alteryx | TRIFACTA

# Wrangle Reference Guide

**Version: 9.7**
**Doc Build Date: 12/31/2022**

## Copyright and Trademark Notices

## Credits

For third-party license information, please select **About Trifacta** from the Resources menu.

See also Third-Party License Information.

# Wrangle Language

**Contents:**

- *Wrangle vs. SQL*
- *Wrangle Syntax*
  - *Common Parameters*
  - *Flow parameters*
  - *Parameter Inputs*
- *Interactions between Wrangle and the Application*
- *Transformation*
- *Functions*
  - *Function input parameters*
  - *Function categories*
- *Operator Categories*
- *Transforms*
- *Documentation*
- *All Topics*

---

Wrangle  is the domain-specific language used to build transformation recipes in  Designer Cloud Powered by Trifacta® Enterprise Edition.

A  **Wrangle  recipe** is a sequence of transformation steps applied to your dataset in order to produce your results.

**Transform and transformation:**

- A **transform** is a single action applied to a dataset. A transform is part of the underlying  Wrangle . Transforms are not directly accessible to users.
- A **transformation** is a user-facing action that you can apply to your dataset through the Transformer page. A transformation is typically a use-specific or more sophisticated manifestation of a transform.

  > **NOTE:** Transforms are a part of the underlying language, which is not directly accessible to users. This content is maintained for reference purposes only. For more information on the user-accessible equivalent to transforms, see *Transformation Reference*.

  > **Tip:** Except for the reference documentation for individual transforms, the language documentation references transformations that you can apply through the Transformer page.

For most of these actions, you can pass one or more **parameters** to define the context (columns, rows, or conditions).

**Function:**

Some parameters accept one or more functions. A **function** is a computational action performed on one or more columns of data in your dataset.

When you select suggestions in the Transformer Page, your selection is converted into a transformation that you can add to your recipe.

> **Tip:** Where possible, you should make selections in the data grid to build transformation steps. These selections prompt a series of cards to be displayed. You can select different cards to specify a basic transformation for your selected data, choose a variant of that transformation, and then modify the underlying Wrangle recipe as necessary.

## Wrangle vs. SQL

> **NOTE:** Wrangle is not SQL. It is a proprietary language of data transformation, purpose-built for Designer Cloud Powered by Trifacta Enterprise Edition.

While there are some overlaps between Wrangle and SQL, here are the key distinctions:

- **Wrangle** is a proprietary language designed for data transformation. Every supported transformation is designed to make changes to a dataset. It cannot be used to read from or write to a datastore.
  - Users interact with Wrangle exclusively through the Designer Cloud application . There is no direct interaction with the language.
- **SQL** (Structured Query Language) is designed for querying, transforming, and writing for relational datasources. It cannot be applied to file-based datasets.
  - SQL cannot be used to transform data in Designer Cloud Powered by Trifacta Enterprise Edition.

## Wrangle Syntax

Wrangle transforms follow this general syntax:

```
(transform) param1:(expression) param2:(expression)
```

| Transform Element | Description |
|---|---|
| transform | In Wrangle , a transform (or verb) is a single keyword that identifies the type of change you are applying to your dataset.<br><br>A transform is always the first keyword in a recipe step. Details are below.<br><br>> **NOTE:** Transforms are a part of the underlying language, which is not directly accessible to users. This content is maintained for reference purposes only. For more information on the user-accessible equivalent to transforms, see *Transformation Reference*.<br><br>The other elements in each step are contextual parameters for the transform. Some transforms do not require parameters. |
| parameter 1:,parame ter2: | Additional parameters may be optional or required for any transform.<br><br>> **NOTE:** A parameter is always followed by a colon. A parameter may appear only one time in a transform step. |

### Common Parameters

Depending on the transform, one or more of `value`, `col`, and `row` parameters may be used. For example, the `set` transform can use all three or just `value` and `col`.

| Transform Element | Description |
|---|---|
|  |  |

| | |
|---|---|
| `value:` | When present, the `value` parameter defines the expression that creates the output value or values stored when the transform is executed.<br><br>An expression can contain combinations of the following:<br><br>• **Functions** apply computations or evaluations of source data, the outputs of which can become inputs to the column. Sources may be constants or column references. A function reference is always followed by brackets (), even if it takes no parameters. See below.<br>• **Operators** are single-character representations of numeric functions, comparisons, or logical operators. For example, the plus sign (+) is the operator for the add function. See below.<br>• **Constants** can be quoted string literals (`'mystring'`), Integer values (`1001`), Decimal values (`1001.01`), Boolean values (`true` or `false`) or patterns.<br><br>For more information on Patterns , see *Text Matching*. |
| `col:` | When present, the `col` parameter identifies the name of the column or columns to which the transform is applied.<br><br>Some transforms may support multiple columns as a list, as a range of columns (e.g., `column1~column5`), or all columns in the dataset (using wildcard indicator, `col: *`). |
| `row:` | When present, the `row` parameter defines the expression to evaluate to determine the rows on which to perform the transform. If the row expression evaluates to `true` for a row, the transform is performed on the row. |
| `group:` | For aggregating transforms, such as `window`, `pivot`, and `derive`, the `group` parameter enables you to calculate aggregation functions within a group value. For example, you can sum sales for each rep by applying `group:repName` to your transformation.<br><br>> **NOTE:** Transforms that use the `group` parameter can result in non-deterministic re-ordering in the data grid. However, you should apply the `group` parameter, particularly on larger datasets, or your job may run out of memory and fail. To enforce row ordering, you can use the `sort` transform. For more information, see *Sort Transform*. |
| `order:` | For aggregating transforms, such as `window` , `pivot` , and `derive` , the `order` parameter can be used to specify the column by which the transform results are sorted. In the previous example, you might choose to sort your sum of sales calculation by state: `order:State`. |

### Flow parameters

At the flow level, you can define parameters that can be referenced in your recipe steps.

In the following transformation, the flow parameter `currentDiscount` is invoked in the step to yield the discounted cost.

| | |
|---|---|
| **Transformation Name** | `New formula` |
| **Parameter: Formula type** | `Single row formula` |
| **Parameter: Formula** | `TotalCost * (1/${currentDiscount})` |
| **Parameter: New column name** | `discountedCost` |

Flow parameters are referenced in your steps in the following format:

```
${MyRecipeParameter}
```

For more information, see *Create Flow Parameter*.

## Parameter Inputs

The following types of parameter inputs may be referenced in a transform's parameters.

Other Trifacta data types can be referenced as column references. For literal values of these data types, you can insert them into your expressions as strings. Transforms cause the resulting values to be re-inferred for their data type.

### Column reference

A reference to the values stored in a column in your dataset. Columns can be referenced by the plain-text value for the column name.

**Example:** `value` parameter references the `myCol` column.

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | myCol |
| Parameter: New column name | 'myNewCol' |

Column names with spaces or special characters in a transformation must be wrapped by curly braces.

**Example:** Below, `srcColumn` is renamed to `src Column`, which requires no braces because the new name is captured as a string literal:

| Transformation Name | Rename columns |
|---|---|
| Parameter: Option | Manual rename |
| Parameter: Column | srcColumn |
| Parameter: New column name | src Column |

> **NOTE:** Current column names that have a space in them must be bracketed in curly braces. The above column name reference is the following: `{src Column}`.

### Functions

Some parameters accept functions as inputs. Where values or formulas are calculated, you can reference one of the dozens of functions available in Wrangle .

**Example:**

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | MULTIPLY(3,2) |
| Parameter: New column name | 'six' |

**Metadata variables**

Wrangle supports the use of variable references to aspects of the source data or dataset. In the following example, the ABS function is applied to each column in a set of them using the `$col` reference.

| Transformation Name | Edit column with formula |
|---|---|
| **Parameter: Columns** | val1,val2 |
| **Parameter: Formula** | ABS($col) |

`$col` returns the value from the current column for the row under evaluation.For more information on these variables, see *Source Metadata References*.

**Nested expressions**

Individual parameters within a function can be computed expressions themselves. These nested expressions can be calculated using constants, other functions, and column references.

**Example:** Computes a column whose only value is ten divided by three, rounded to the nearest integer (3):

| Transformation Name | New formula |
|---|---|
| **Parameter: Formula type** | Single row formula |
| **Parameter: Formula** | ROUND(DIVIDE(10,3),0) |
| **Parameter: New column name** | 'three' |

**Integer**

A valid integer value within the accepted range of values for the Integer datatype.

**Example:** Generates a column called, `my13` which is the sum of the Integer values `5` and `8`:

| Transformation Name | New formula |
|---|---|
| **Parameter: Formula type** | Single row formula |
| **Parameter: Formula** | (5 + 8) |
| **Parameter: New column name** | 'my13' |

**Decimal**

A valid floating point value within the accepted range of values for the Decimal datatype.

**Example:** Generates a column of values that computes the approximate circumference of the values in the `diameter` column:

| Transformation Name | New formula |
|---|---|
| **Parameter: Formula type** | Single row formula |
| **Parameter: Formula** | (3.14159 * diameter) |

| Parameter: New column name | `'circumference'` |
| --- | --- |

**Boolean**

A `true` or `false` value.

**Example:** If the value in the `order` column is more than 1,000,000, then the value in the `bigOrder` column is `true`.

| Transformation Name | `New formula` |
| --- | --- |
| Parameter: Formula type | `Single row formula` |
| Parameter: Formula | `IF(order > 1000000, true, false)` |
| Parameter: New column name | `'bigOrder'` |

**String**

A string literal value is the baseline datatype. String literals must be enclosed in single quotes.

**Example:** Creates a column called, `StringCol` containing the value `myString`.

| Transformation Name | `New formula` |
| --- | --- |
| Parameter: Formula type | `Single row formula` |
| Parameter: Formula | `'myString'` |
| Parameter: New column name | `'StringCol'` |

**Trifacta pattern**

Designer Cloud Powered by Trifacta Enterprise Edition supports a special syntax, which simplifies the generation of matching patterns for string values.

Patterns must be enclosed in accent marks ( `` `MyPattern` ``). For more information on Patterns , see *Text Matching.***Example:** Extracts up to 10 values from the `MyData` column that match the basic pattern for social security numbers (`XXX-XX-XXXX`):

| Transformation Name | `Extract text or pattern` |
| --- | --- |
| Parameter: Column to extract from | `MyData` |
| Parameter: Option | `Custom text or pattern` |
| Parameter: Text to extract | `` `%{3}-%{2}-%{4}` `` |
| Parameter: Number of matches to extract | `10` |

**Regular expression**

Regular expressions are a common standard for defining matching patterns. Regex is a very powerful tool but can be easily misconfigured.

Regular expressions must be enclosed in slashes ( /MyPattern/ ).

**Example:** Deletes all two-digit numbers from the `qty` column:

| Transformation Name | Replace text or pattern |
|---|---|
| Parameter: Column | qty |
| Parameter: Find | /^\d$|^\d\d$/ |
| Parameter: Replace with | '' |
| Parameter: Match all occurrences | true |

**Datetime**

A valid date or time value that matches the requirements of the Datetime datatype.

Datetime values can be formatted with specific formatting strings. See *DATEFORMAT Function*.

**Example:** Generates a new column containing the values from the `myDate` column reformatted in `yyyymmdd` format:

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | DATEFORMAT(myDate, 'yyyymmdd') |

**Array**

A valid array of values matching the Array data type.

**Example:**

```
[0,1,2,3,4,5,6,7,8]
```

**Example:** Generates a column with the number of elements in the listed array (7):

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | ARRAYLEN('["red", "orange", "yellow", "green", "blue", "indigo", "violet"]') |

**Object**

A valid set of values matching the Object data type.

**Example:**

```
{"brand":"Subaru","model":"Impreza","color","green"}
```

**Example:** Generates separate columns for each of the specified keys in the object ( `brand`, `model`, `color`), containing the corresponding value for each row:

| Transformation Name | Unnest Objects into columns |
| --- | --- |
| Parameter: Column | myCol |
| Parameter: Paths to elements | 'brand','model','color' |

For more information on these data types, see *Supported Data Types*.

## Interactions between Wrangle and the Application

1. As you build Wrangle steps in the Transform Builder, your syntax is validated for you. You cannot add steps containing invalid syntax.
   a. Error messages are reported back to the application, so you can make immediate modifications to correct the issue.
   b. Type-ahead support can provide guidance to the supported transforms, functions, and column references.
2. When you have entered a valid transform step, the results are previewed for you in the data grid.
   a. This preview is generated by applying the transformation to the sample in the data grid.

   > **NOTE:** The generated output applies only to the values displayed in the data grid. The function is applied across the entire dataset only during job execution.

   b. If the previewed transformation is invalid, the data grid is grayed out.
3. When you add the transformation to your recipe:
   a. It is applied to the sample in the application, and the data grid is updated to the current state.
   b. Column histograms are updated with new values and counts.
   c. Column data types may be re-inferred for affected columns.
4. Making changes:
   a. You can edit any transformation step in your recipe whenever needed.
      i. When you edit a transformation step in your recipe, the context of the data grid is changed to display the state of your data up to the point of previewing the step you're editing.
      ii. All subsequent steps are still part of the recipe, but they are not applied to the sample yet.
      iii. You can insert recipe steps between existing steps.
   b. When you delete a recipe step, the state remains at the point where the step was removed.
      i. You can insert a new step if needed.
      ii. When you complete your edit, select the final step of the recipe, which displays the results of all of your transformation steps in the data grid. Your changes may cause some recipe steps to become invalid.

For more information on these areas of the Designer Cloud application , see *Transformer Page*.

## Transformation

A transformation is an action for which you can browse or search through the Transform Builder in the Transformer page. When specified and added to the recipe, these sometimes complex actions are rendered in the recipe as steps using the underlying transforms of the language.

> **Tip:** Through transformations, Designer Cloud Powered by Trifacta Enterprise Edition guides you through creation of more sophisticated steps that would be difficult to create in raw Wrangle .

For more information on the list of available transformations, see *Transformation Reference*.

For more information on creating transformation steps in the Transformer page, see *Transform Builder*.

## Functions

A **function** is an action that is applied to a set of values as part of a transform step. Functions can apply to the values in a transform for specific data types, such as strings, or to types of transforms, such as aggregate and window function categories. A function cannot be applied to data without a transform.

### Function input parameters

Below, function inputs are listed in increasing order of generality.

> **NOTE:** A function cannot take a higher-order parameter input type without taking the lower parameter input types. For example, a function cannot take a nested function as an input if it does not accept a literal value, too.

| Order | Parameter input type | Example |
|-------|---------------------|---------|
| 1 | literal | `FUNCTION('my input')` |
| 2 | column | `FUNCTION(myColumnOfValues)` |
| 3 | function | `FUNCTION(SUM(MyCol))` |

### Function categories

| Function Category | Description |
|-------------------|-------------|
| *Aggregate Functions* | These functions are used to perform aggregation calculations on your data, such as sum, mean, and standard deviation. |
| *Comparison Functions* | Comparison functions enable evaluation between two data elements, which are typically nested (Object or Array) elements. |
| *Math Functions* | Perform computations on your data using a variety of math functions and numeric operators. |
| *Trigonometry Functions* | Calculate standard trigonometry functions as well as arc versions of them. |
| *Date Functions* | Use these functions to extract data from or perform operations on objects of Datetime data type. |
| *String Functions* | Manipulate strings, including finding sub-strings within a string. |
| *Nested Functions* | These functions are designed specifically to assist in wrangling nested data, such as Objects, Arrays, or JSON elements. |
| *Type Functions* | Use the Type functions to identify valid, missing, mismatched, and null values. |
| *Window Functions* | The Window functions enable you to perform calculations on relative windows of data within your dataset. |
| *Other Functions* | Miscellaneous functions that do not fit into the other categories |

## Operator Categories

An **operator** is a single character that represents an arithmetic function. For example, the Plus sign (+) represents the add function.

| Operator Category | Description |
|---|---|
| *Logical Operators* | and, or, and not operators |
| *Numeric Operators* | Add, subtract, multiply, and divide |
| *Comparison Operators* | Compare two values with greater than, equals, not equals, and less than operators |
| *Ternary Operators* | Use ternary operators to create if/then/else logic in your transforms. |

## Transforms

A **transform**, or verb, is an action applied to rows or columns of your data. Transforms are the essential set of changes that you can apply to your dataset.

> **NOTE:** Transforms are a part of the underlying language, which is not directly accessible to users. This content is maintained for reference purposes only. For more information on the user-accessible equivalent to transforms, see *Transformation Reference*.

Transforms are described in the Language Appendices. For more information, see *Transforms*.

## Documentation

Documentation for Wrangle is also available through Designer Cloud Powered by Trifacta Enterprise Edition. Select **Resources menu > Documentation**.

> **Tip:** When searching for examples of functions, try using the following form for your search terms within the Product Documentation site:
>
> * Functions: `wrangle_function_NameOfFunction`

## All Topics

# Aggregate Functions

Aggregate functions perform a computation against a set of values to generate a single result. For example, you could use an aggregate function to compute the average (mean) order over a period of time. Aggregations can be applied as standard functions or used as part of a transformation step to reshape the data.

**Aggregate across an entire column:**

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | average(Scores) |

**Output:** Generates a new column containing the average of all values in the `Scores` column.

| Transformation Name | Pivot columns |
|---|---|
| Parameter: Values | average(Score) |
| Parameter: Max number of columns to create | 1 |

**Output:** Generates a single-column table with a single value, which contains the average of all values in the `Scores` column. The limit defines the maximum number of columns that can be generated.

> **NOTE:** When aggregate functions are applied as part of a pivot transformation, they typically involve multiple parameters as part of an operation to reshape the dataset. See below.

**Aggregate across groups of values within a column:**

Aggregate functions can be used with the pivot transformation to change the structure of your data. Example:

| Transformation Name | Pivot columns |
|---|---|
| Parameter: Row labels | StudentId |
| Parameter: Values | average(Score) |
| Parameter: Max number of columns to create | 1 |

In the above instance, the resulting dataset contains two columns:

- `studentId` - one row for each distinct student ID value
- `average_Scores` - average score by each student (`studentId`)

> **NOTE:** You cannot use aggregate functions inside of conditionals that evaluate to `true` or `false`.

A pivot transformation can include multiple aggregate functions and group columns from the pre-aggregate dataset. See *Pivot Transform*.

> **NOTE:** Null values are ignored as inputs to these functions.

These aggregate functions are available:

# ANY Function

Extracts a non-null and non-missing value from a specified column. If all values are missing or null, the function returns a null value.

This function is intended to be used as part an aggregation to return any single value. When run at scale, there is some randomness to the value that is returned from the aggregated groupings, although randomness in not guaranteed.

In a flat aggregation, in which no aggregate function is applied, it selects the first value that it can retrieve from a column, which is the first value. This function has limited value outside of an aggregation. See *Pivot Transform*.

Input column might be of Integer, Decimal, String, Object, or Array type.

**Wrangle vs. SQL:** This function is part of Wrangle , a proprietary data transformation language. Wrangle is not SQL. For more information, see *Wrangle Language*.

## Basic Usage

```
any(myRating)
```

**Output:** Returns a single value from the `myRating` column.

## Syntax and Arguments

```
any(function_col_ref) [group:group_col_ref] [limit:limit_count]
```

| Argument | Required? | Data Type | Description |
|---|---|---|---|
| function_col_ref | Y | string | Name of column to which to apply the function |

For more information on syntax standards, see *Language Documentation Syntax Notes*.

For more information on the `group` and `limit` parameters, see *Pivot Transform*.

### function_col_ref

Name of the column from which to extract a value based on the grouping.

- Literal values are not supported as inputs.
- Multiple columns and wildcards are not supported.

**Usage Notes:**

| Required? | Data Type | Example Value |
|---|---|---|
| Yes | String (column reference) | `myValues` |

## Examples

### Example - Aggregating on one customer per month

You want to do some sampling of customer orders on a monthly basis. For your sample, you want to select the sum of orders for one customer each month.

**Source:**

Here are the orders for 1Q 2015:

| OrderId | Date | CustId | Qty |
|---------|---------|--------|-----|
| 1001 | 1/8/15 | C0001 | 12 |
| 1002 | 2/12/15 | C0002 | 65 |
| 1003 | 1/16/15 | C0004 | 23 |
| 1004 | 1/31/15 | C0002 | 92 |
| 1005 | 2/2/15 | C0005 | 56 |
| 1006 | 3/2/15 | C0006 | 83 |
| 1007 | 3/16/15 | C0005 | 62 |
| 1008 | 2/21/15 | C0002 | 43 |
| 1009 | 3/28/15 | C0001 | 86 |

**Transformation:**

To aggregate this date by month, you must extract the month value from the `Date` column:

| Transformation Name | New formula |
|---------------------|-------------|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | dateformat(Date, 'MMM') |
| Parameter: New column name | 'month_Date' |

You should now have a new column with three-letter month abbreviations. You can use the following aggregation to gather the sum of one customer's orders for each month:

| Transformation Name | Pivot columns |
|---------------------|---------------|
| Parameter: Row labels | month_Date |
| Parameter: Values | any(CustId),sum(Qty) |
| Parameter: Max columns to create | 1 |

**Results:**

| month_Date | any_CustId | sum_Qty |
|------------|------------|---------|
| Jan | C0001 | 127 |

| Feb | C0002 | 164 |
|-----|-------|-----|
| Mar | C0006 | 211 |

# ANYIF Function

**Contents:**

Selects a single non-null value from rows in each group that meet a specific condition.

> **NOTE:** When added to a transformation, this function is applied to the current sample. If you change your sample or run the job, the computed values for this function are updated. Transformations that change the number of rows in subsequent recipe steps do not affect the values computed for this step.

To perform a simple counting of non-nulls without conditionals, use the `ANY` function. See *ANY Function*.

**Wrangle vs. SQL:** This function is part of Wrangle , a proprietary data transformation language. Wrangle is not SQL. For more information, see *Wrangle Language*.

## Basic Usage

```
anyif(custId, donation = 10000)
```

**Output:** Returns a single value from `custId` when the `donation` value is greater than `10000`.

## Syntax and Arguments

```
anyif(col_ref, test_expression) [group:group_col_ref] [limit:limit_count]
```

| Argument | Required? | Data Type | Description |
|---|---|---|---|
| col_ref | Y | string | Reference to the column you wish to evaluate. |
| test_expression | Y | string | Expression that is evaluated. Must resolve to `true` or `false` |

For more information on syntax standards, see *Language Documentation Syntax Notes*.

For more information on the `group` and `limit` parameters, see *Pivot Transform*.

**col_ref**

Name of the column whose values you wish to use in the calculation. Column must be a numeric (Integer or Decimal) type.

**Usage Notes:**

| Required? | Data Type | Example Value |
|---|---|---|
| Yes | String that corresponds to the name of the column | `myValues` |

### test_expression

This parameter contains the expression to evaluate. This expression must resolve to a Boolean (`true` or `false`) value.

**Usage Notes:**

| Required? | Data Type | Example Value |
|---|---|---|
| Yes | String expression that evaluates to `true` or `false` | `(LastName == 'Mouse' && FirstName == 'Mickey')` |

## Examples

> **Tip:** For additional examples, see *Common Tasks*.

### Example - ANYIF and LISTIF Functions

This example illustrates you to identify and list all values within a group that meet a specified condition.

**Functions**:

| Item | Description |
|---|---|
| ANYIF Function | Selects a single non-null value from rows in each group that meet a specific condition. |
| LISTIF Function | Returns list of all values in a column for rows that match a specified condition. |
| WEEKDAY Function | Derives the numeric value for the day of the week (`1`, `2`, etc.). Input must be a reference to a column containing Datetime values. |

**Source:**

The following data identifies sales figures by salespeople for a week:

| EmployeeId | Date | Sales |
|---|---|---|
| S001 | 1/23/17 | 25 |
| S002 | 1/23/17 | 40 |
| S003 | 1/23/17 | 48 |
| S001 | 1/24/17 | 81 |
| S002 | 1/24/17 | 11 |
| S003 | 1/24/17 | 25 |
| S001 | 1/25/17 | 9 |
| S002 | 1/25/17 | 40 |
| S003 | 1/25/17 | |

| | | |
|---|---|---|
| S001 | 1/26/17 | 77 |
| S002 | 1/26/17 | 83 |
| S003 | 1/26/17 | |
| S001 | 1/27/17 | 17 |
| S002 | 1/27/17 | 71 |
| S003 | 1/27/17 | 29 |
| S001 | 1/28/17 | |
| S002 | 1/28/17 | |
| S003 | 1/28/17 | 14 |
| S001 | 1/29/17 | 2 |
| S002 | 1/29/17 | 7 |
| S003 | 1/29/17 | 99 |

**Transformation:**

In this example, you are interested in the high performers. A good day in sales is one in which an individual sells more than 80 units. First, you want to identify the day of week:

| | |
|---|---|
| **Transformation Name** | `New formula` |
| **Parameter: Formula type** | `Single row formula` |
| **Parameter: Formula** | `WEEKDAY(Date)` |
| **Parameter: New column name** | `'DayOfWeek'` |

Values greater than 5 in `DayOfWeek` are weekend dates. You can use the following to identify if anyone reached this highwater marker during the workweek (non-weekend):

| | |
|---|---|
| **Transformation Name** | `Pivot columns` |
| **Parameter: Rows labels** | `EmployeeId,Date` |
| **Parameter: Values** | `ANYIF(Sales, (Sales > 80 && DayOfWeek < 6))` |
| **Parameter: Max number of columns to create** | `1` |

Before adding the step to the recipe, you take note of the individuals who reached this mark in the `anyif_Sales` column for special recognition.

Now, you want to find out sales for individuals during the week. You can use the following to filter the data to show only for weekdays:

| | |
|---|---|
| **Transformation Name** | `Pivot columns` |
| **Parameter: Rows labels** | `EmployeeId,Date` |
| **Parameter: Values** | `LISTIF(Sales, 1000, (DayOfWeek < 6))` |

| Parameter: Max number of columns to create | 1 |
|---|---|

To clean up, you might select and replace the following values in the listif_Sales column with empty strings:

```
[ "
" ]
[ ]
```

## Results:

| EmployeeId | Date | listif_Sales |
|---|---|---|
| S001 | 1/23/17 | 25 |
| S002 | 1/23/17 | 40 |
| S003 | 1/23/17 | 48 |
| S001 | 1/24/17 | 81 |
| S002 | 1/24/17 | 11 |
| S003 | 1/24/17 | 25 |
| S001 | 1/25/17 | 40 |
| S002 | 1/25/17 | |
| S003 | 1/25/17 | 66 |
| S001 | 1/26/17 | 77 |
| S002 | 1/26/17 | 83 |
| S003 | 1/26/17 | |
| S001 | 1/27/17 | 17 |
| S002 | 1/27/17 | 71 |
| S003 | 1/27/17 | 29 |
| S001 | 1/28/17 | |
| S002 | 1/28/17 | |
| S003 | 1/28/17 | |
| S001 | 1/29/17 | |
| S002 | 1/29/17 | |
| S003 | 1/29/17 | |

# APPROXIMATEMEDIAN Function

**Contents:**

Computes the approximate median from all row values in a column or group. Input column can be of Integer or Decimal.

- If a row contains a missing or null value, it is not factored into the calculation. If the entire column contains no values, the function returns a null value.
- When used in a `pivot` transform, the function is computed for each instance of the value specified in the `group` parameter. See *Pivot Transform*.
- The approximate percentile functions utilize a different algorithm for efficiently estimating quantiles for streaming and distributed processing, depending on the running environment where the function is computed.

> **Tip:** Approximation functions are suitable for larger datasets. As the number of rows increases, accuracy and calculation performance improves for these functions.

- For an exact calculation of this function, see *MEDIAN Function*.

**Wrangle vs. SQL:** This function is part of Wrangle , a proprietary data transformation language. Wrangle is not SQL. For more information, see *Wrangle Language*.

## Basic Usage

```
approximatemedian(myRating)
```

**Output:** Returns the approximate median of the values in the `myRating` column.

## Syntax and Arguments

```
approximatemedian(function_col_ref) [group:group_col_ref] [limit:limit_count]
```

| Argument | Required? | Data Type | Description |
|---|---|---|---|
| function_col_ref | Y | string | Name of column to which to apply the function |
| dec_error_bound | N | decimal | Error factor for computing approximations. Decimal value represents error factor as a percentage (0.4 is 0.4%). |

For more information on the `group` and `limit` parameters, see *Pivot Transform*.

For more information on syntax standards, see *Language Documentation Syntax Notes*.

**function_col_ref**

Name of the column the values of which you want to calculate the median. Column must contain Integer or Decimal values.

- Literal values are not supported as inputs.
- Multiple columns and wildcards are not supported.

**Usage Notes:**

| Required? | Data Type | Example Value |
|-----------|-----------|---------------|
| Yes | String (column reference) | `myValues` |

**dec_error_bound**

As needed, you can insert an error boundary factor as a parameter into the computation of this approximate value.

> **NOTE:** This value is not applicable to jobs executed on the Trifacta Photon running environment.

- This value must be a Decimal literal value.
- This decimal value represents the percentage error factor. By default, this value is `0.5` (0.5%).

**Usage Notes:**

| Required? | Data Type | Example Value |
|-----------|-----------|---------------|
| No | Decimal (literal) | `0.01` |

Examples

> **Tip:** For additional examples, see *Common Tasks*.

**Example - Percentile functions**

This example illustrates you to apply percentile functions.

**Functions**:

| Item | Description |
|------|-------------|
| MEDIAN Function | Computes the median from all row values in a column or group. Input column can be of Integer or Decimal. |
| PERCENTILE Function | Computes a specified percentile across all row values in a column or group. Input column can be of Integer or Decimal. |
| QUARTILE Function | Computes a specified quartile across all row values in a column or group. Input column can be of Integer or Decimal. |

| | |
|---|---|
| APPROXIMATEMEDIAN Function | Computes the approximate median from all row values in a column or group. Input column can be of Integer or Decimal. |
| APPROXIMATEPERCENTIL E Function | Computes an approximation for a specified percentile across all row values in a column or group. Input column can be of Integer or Decimal. |
| APPROXIMATEQUARTILE Function | Computes an approximation for a specified quartile across all row values in a column or group. Input column can be of Integer or Decimal. |

**Source:**

The following table lists each student's height in inches:

| Student | Height |
|---|---|
| 1 | 64 |
| 2 | 65 |
| 3 | 63 |
| 4 | 64 |
| 5 | 62 |
| 6 | 66 |
| 7 | 66 |
| 8 | 65 |
| 9 | 69 |
| 10 | 66 |
| 11 | 73 |
| 12 | 69 |
| 13 | 69 |
| 14 | 61 |
| 15 | 64 |
| 16 | 61 |
| 17 | 71 |
| 18 | 67 |
| 19 | 73 |
| 20 | 66 |

**Transformation:**

Use the following transformations to calculate the median height in inches, a specified percentile and the first quartile.

- The first function uses a precise algorithm which can be slow to execute across large datasets.
- The second function uses an appropriate approximation algorithm, which is much faster to execute across large datasets.
  - These approximate functions can use an error boundary parameter, which is set to `0.4` (0.4%) across all functions.

**Median:** This transformation calculates the median value, which corresponds to the 50th percentile.

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | median(heightIn) |
| Parameter: New column name | 'medianIn' |

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | approximatemedian(heightIn, 0.4) |
| Parameter: New column name | 'approxMedianIn' |

**Percentile:** This transformation calculates the 68th percentile.

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | percentile(heightIn, 68, linear) |
| Parameter: New column name | 'percentile68In' |

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | approximatepercentile(heightIn, 68, 0.4) |
| Parameter: New column name | 'approxPercentile68In' |

**Quartile:** This transformation calculates the first quartile, which corresponds to the 25th percentile.

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | quartile(heightIn, 1, linear) |
| Parameter: New column name | 'percentile25In' |

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | approximatequartile(heightIn, 1, 0.4) |
| Parameter: New column name | 'approxPercentile25In' |

**Results:**

| studentId | heightIn | approxPercentile25In | percentile25In | approxPercentile68In | percentile68In | approxMedianIn | |
|-----------|----------|----------------------|----------------|----------------------|----------------|----------------|---|
| 1 | 64 | 64 | 64 | 67.1 | 66.92 | 66 | 6 |
| 2 | 65 | 64 | 64 | 67.1 | 66.92 | 66 | 6 |
| 3 | 63 | 64 | 64 | 67.1 | 66.92 | 66 | 6 |
| 4 | 64 | 64 | 64 | 67.1 | 66.92 | 66 | 6 |
| 5 | 62 | 64 | 64 | 67.1 | 66.92 | 66 | 6 |
| 6 | 66 | 64 | 64 | 67.1 | 66.92 | 66 | 6 |
| 7 | 66 | 64 | 64 | 67.1 | 66.92 | 66 | 6 |
| 8 | 65 | 64 | 64 | 67.1 | 66.92 | 66 | 6 |
| 9 | 69 | 64 | 64 | 67.1 | 66.92 | 66 | 6 |
| 10 | 66 | 64 | 64 | 67.1 | 66.92 | 66 | 6 |
| 11 | 73 | 64 | 64 | 67.1 | 66.92 | 66 | 6 |
| 12 | 69 | 64 | 64 | 67.1 | 66.92 | 66 | 6 |
| 13 | 69 | 64 | 64 | 67.1 | 66.92 | 66 | 6 |
| 14 | 61 | 64 | 64 | 67.1 | 66.92 | 66 | 6 |
| 15 | 64 | 64 | 64 | 67.1 | 66.92 | 66 | 6 |
| 16 | 61 | 64 | 64 | 67.1 | 66.92 | 66 | 6 |
| 17 | 71 | 64 | 64 | 67.1 | 66.92 | 66 | 6 |
| 18 | 67 | 64 | 64 | 67.1 | 66.92 | 66 | 6 |
| 19 | 73 | 64 | 64 | 67.1 | 66.92 | 66 | 6 |
| 20 | 66 | 64 | 64 | 67.1 | 66.92 | 66 | 6 |

# APPROXIMATEPERCENTILE Function

**Contents:**

Computes an approximation for a specified percentile across all row values in a column or group. Input column can be of Integer or Decimal.

- If a row contains a missing or null value, it is not factored into the calculation. If the entire column contains no values, the function returns a null value.
- When used in a `pivot` transform, the function is computed for each instance of the value specified in the `group` parameter. See *Pivot Transform*.
- The approximate percentile functions utilize a different algorithm for efficiently estimating quantiles for streaming and distributed processing, depending on the running environment where the function is computed.

> **Tip:** Approximation functions are suitable for larger datasets. As the number of rows increases, accuracy and calculation performance improves for these functions.

- For an exact calculation of this function, see *PERCENTILE Function*.

**Wrangle vs. SQL:** This function is part of Wrangle , a proprietary data transformation language. Wrangle is not SQL. For more information, see *Wrangle Language*.

## Basic Usage

```
approximatepercentile(myScores, 25)
```

**Output:** Computes the approximate value that is at the 25th percentile across all values in the `myScores` column.

## Syntax and Arguments

```
approximatepercentile(function_col_ref,num_percentile) [group:group_col_ref] [limit:
limit_count]
```

| Argument | Required? | Data Type | Description |
|---|---|---|---|
| function_col_ref | Y | string | Name of column to which to apply the function |
| num_percentile | Y | integer | Integer value between 1-100 of the percentile to compute |

| dec_error_bo und | N | decimal | Error factor for computing approximations. Decimal value represents error factor as a percentage (0.4 is 0.4%). |
|---|---|---|---|

For more information on the `group` and `limit` parameters, see *Pivot Transform*.

For more information on syntax standards, see *Language Documentation Syntax Notes*.

### function_col_ref

Name of the column the values of which you want to calculate the percentile. Column must contain Integer or Decimal values.

- Literal values are not supported as inputs.
- Multiple columns and wildcards are not supported.

**Usage Notes:**

| Required? | Data Type | Example Value |
|---|---|---|
| Yes | String (column reference) | `precipitationI n` |

### num_percentile

Integer literal value indicating the percentile that you wish to compute. Input value must be between `1` and `100`, inclusive.

- Column or function references are not supported.
- Multiple columns and wildcards are not supported.

**Usage Notes:**

| Required? | Data Type | Example Value |
|---|---|---|
| Yes | Integer | `55` |

### dec_error_bound

As needed, you can insert an error boundary factor as a parameter into the computation of this approximate value.

> **NOTE:** This value is not applicable to jobs executed on the Trifacta Photon running environment.

- This value must be a Decimal literal value.
- This decimal value represents the percentage error factor. By default, this value is `0.5` (0.5%).

**Usage Notes:**

| Required? | Data Type | Example Value |
|---|---|---|
| No | Decimal (literal) | `0.01` |

## Examples

> **Tip:** For additional examples, see *Common Tasks.*

### Example - Percentile functions

This example illustrates you to apply percentile functions.

**Functions**:

| Item | Description |
|---|---|
| MEDIAN Function | Computes the median from all row values in a column or group. Input column can be of Integer or Decimal. |
| PERCENTILE Function | Computes a specified percentile across all row values in a column or group. Input column can be of Integer or Decimal. |
| QUARTILE Function | Computes a specified quartile across all row values in a column or group. Input column can be of Integer or Decimal. |
| APPROXIMATEMEDIAN Function | Computes the approximate median from all row values in a column or group. Input column can be of Integer or Decimal. |
| APPROXIMATEPERCENTILE Function | Computes an approximation for a specified percentile across all row values in a column or group. Input column can be of Integer or Decimal. |
| APPROXIMATEQUARTILE Function | Computes an approximation for a specified quartile across all row values in a column or group. Input column can be of Integer or Decimal. |

**Source:**

The following table lists each student's height in inches:

| Student | Height |
|---|---|
| 1 | 64 |
| 2 | 65 |
| 3 | 63 |
| 4 | 64 |
| 5 | 62 |
| 6 | 66 |
| 7 | 66 |
| 8 | 65 |
| 9 | 69 |
| 10 | 66 |
| 11 | 73 |
| 12 | 69 |
| 13 | 69 |
| 14 | 61 |
| 15 | 64 |
| 16 | 61 |
| 17 | 71 |
| 18 | 67 |

| | |
|---|---|
| 19 | 73 |
| 20 | 66 |

**Transformation:**

Use the following transformations to calculate the median height in inches, a specified percentile and the first quartile.

- The first function uses a precise algorithm which can be slow to execute across large datasets.
- The second function uses an appropriate approximation algorithm, which is much faster to execute across large datasets.
    - These approximate functions can use an error boundary parameter, which is set to `0.4` (0.4%) across all functions.

**Median:** This transformation calculates the median value, which corresponds to the 50th percentile.

| | |
|---|---|
| **Transformation Name** | `New formula` |
| **Parameter: Formula type** | `Single row formula` |
| **Parameter: Formula** | `median(heightIn)` |
| **Parameter: New column name** | `'medianIn'` |

| | |
|---|---|
| **Transformation Name** | `New formula` |
| **Parameter: Formula type** | `Single row formula` |
| **Parameter: Formula** | `approximatemedian(heightIn, 0.4)` |
| **Parameter: New column name** | `'approxMedianIn'` |

**Percentile:** This transformation calculates the 68th percentile.

| | |
|---|---|
| **Transformation Name** | `New formula` |
| **Parameter: Formula type** | `Single row formula` |
| **Parameter: Formula** | `percentile(heightIn, 68, linear)` |
| **Parameter: New column name** | `'percentile68In'` |

| | |
|---|---|
| **Transformation Name** | `New formula` |
| **Parameter: Formula type** | `Single row formula` |
| **Parameter: Formula** | `approximatepercentile(heightIn, 68, 0.4)` |
| **Parameter: New column name** | `'approxPercentile68In'` |

**Quartile:** This transformation calculates the first quartile, which corresponds to the 25th percentile.

| | |
|---|---|
| **Transformation Name** | `New formula` |

| Parameter: Formula type | Single row formula |
|---|---|
| Parameter: Formula | quartile(heightIn, 1, linear) |
| Parameter: New column name | 'percentile25In' |

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | approximatequartile(heightIn, 1, 0.4) |
| Parameter: New column name | 'approxPercentile25In' |

**Results:**

| studentId | heightIn | approxPercentile25In | percentile25In | approxPercentile68In | percentile68In | approxMedianIn | |
|---|---|---|---|---|---|---|---|
| 1 | 64 | 64 | 64 | 67.1 | 66.92 | 66 | 6 |
| 2 | 65 | 64 | 64 | 67.1 | 66.92 | 66 | 6 |
| 3 | 63 | 64 | 64 | 67.1 | 66.92 | 66 | 6 |
| 4 | 64 | 64 | 64 | 67.1 | 66.92 | 66 | 6 |
| 5 | 62 | 64 | 64 | 67.1 | 66.92 | 66 | 6 |
| 6 | 66 | 64 | 64 | 67.1 | 66.92 | 66 | 6 |
| 7 | 66 | 64 | 64 | 67.1 | 66.92 | 66 | 6 |
| 8 | 65 | 64 | 64 | 67.1 | 66.92 | 66 | 6 |
| 9 | 69 | 64 | 64 | 67.1 | 66.92 | 66 | 6 |
| 10 | 66 | 64 | 64 | 67.1 | 66.92 | 66 | 6 |
| 11 | 73 | 64 | 64 | 67.1 | 66.92 | 66 | 6 |
| 12 | 69 | 64 | 64 | 67.1 | 66.92 | 66 | 6 |
| 13 | 69 | 64 | 64 | 67.1 | 66.92 | 66 | 6 |
| 14 | 61 | 64 | 64 | 67.1 | 66.92 | 66 | 6 |
| 15 | 64 | 64 | 64 | 67.1 | 66.92 | 66 | 6 |
| 16 | 61 | 64 | 64 | 67.1 | 66.92 | 66 | 6 |
| 17 | 71 | 64 | 64 | 67.1 | 66.92 | 66 | 6 |
| 18 | 67 | 64 | 64 | 67.1 | 66.92 | 66 | 6 |
| 19 | 73 | 64 | 64 | 67.1 | 66.92 | 66 | 6 |
| 20 | 66 | 64 | 64 | 67.1 | 66.92 | 66 | 6 |

# APPROXIMATEQUARTILE Function

**Contents:**

- *Basic Usage*
- *Syntax and Arguments*
    - *function_col_ref*
    - *num_quartile*
    - *dec_error_bound*
- *Examples*
    - *Example - Percentile functions*

---

Computes an approximation for a specified quartile across all row values in a column or group. Input column can be of Integer or Decimal.

- If a row contains a missing or null value, it is not factored into the calculation. If the entire column contains no values, the function returns a null value.
- When used in a `pivot` transform, the function is computed for each instance of the value specified in the `group` parameter. See *Pivot Transform*.
- The approximate percentile functions utilize a different algorithm for efficiently estimating quantiles for streaming and distributed processing, depending on the running environment where the function is computed.

> **Tip:** Approximation functions are suitable for larger datasets. As the number of rows increases, accuracy and calculation performance improves for these functions.

- For an exact calculation of this function, see *QUARTILE Function*.

Quartiles are computed as follows:

| Quartile | Description |
| --- | --- |
| 0 | Minimum value |
| 1 | 25th percentile |
| 2 | Median value |
| 3 | 75th percentile and higher |

**Wrangle vs. SQL:** This function is part of Wrangle , a proprietary data transformation language. Wrangle is not SQL. For more information, see *Wrangle Language*.

## Basic Usage

```
<span>approximatequartile</span>(myScores, 3)
```

**Output:** Computes the approximate value that is at the 3rd quartile across all values in the `myScores` column.

## Syntax and Arguments

```
approximatequartile(function_col_ref,num_quartile) [group:group_col_ref] [limit:
limit_count]
```

| Argument | Required? | Data Type | Description |
|---|---|---|---|
| function_col_r ef | Y | string | Name of column to which to apply the function |
| num_quartile | Y | integer | Integer value (0-3) of the quartile to compute |
| dec_error_bo und | N | decimal | Error factor for computing approximations. Decimal value represents error factor as a percentage (0.4 is 0.4%). |

For more information on the `group` and `limit` parameters, see *Pivot Transform*.

For more information on syntax standards, see *Language Documentation Syntax Notes*.

### function_col_ref

Name of the column the values of which you want to calculate the quartile. Column must contain Integer or Decimal values.

- Literal values are not supported as inputs.
- Multiple columns and wildcards are not supported.

### Usage Notes:

| Required? | Data Type | Example Value |
|---|---|---|
| Yes | String (column reference) | `precipitationI n` |

### num_quartile

Integer literal value indicating the quartile that you wish to compute. Input value must be between `0` and `3`, inclusive.

- Column or function references are not supported.
- Multiple columns and wildcards are not supported.

### Usage Notes:

| Required? | Data Type | Example Value |
|---|---|---|
| Yes | Integer | 3 |

### dec_error_bound

As needed, you can insert an error boundary factor as a parameter into the computation of this approximate value.

> **NOTE:** This value is not applicable to jobs executed on the Trifacta Photon running environment.

-
  This value must be a Decimal literal value.
- This decimal value represents the percentage error factor. By default, this value is `0.5` (0.5%).

### Usage Notes:

| Required? | Data Type | Example Value |
|-----------|-----------|---------------|
| No | Decimal (literal) | `0.01` |

## Examples

> **Tip:** For additional examples, see *Common Tasks*.

### Example - Percentile functions

This example illustrates you to apply percentile functions.

**Functions**:

| Item | Description |
|------|-------------|
| MEDIAN Function | Computes the median from all row values in a column or group. Input column can be of Integer or Decimal. |
| PERCENTILE Function | Computes a specified percentile across all row values in a column or group. Input column can be of Integer or Decimal. |
| QUARTILE Function | Computes a specified quartile across all row values in a column or group. Input column can be of Integer or Decimal. |
| APPROXIMATEMEDIAN Function | Computes the approximate median from all row values in a column or group. Input column can be of Integer or Decimal. |
| APPROXIMATEPERCENTILE Function | Computes an approximation for a specified percentile across all row values in a column or group. Input column can be of Integer or Decimal. |
| APPROXIMATEQUARTILE Function | Computes an approximation for a specified quartile across all row values in a column or group. Input column can be of Integer or Decimal. |

**Source:**

The following table lists each student's height in inches:

| Student | Height |
|---------|--------|
| 1 | 64 |
| 2 | 65 |
| 3 | 63 |
| 4 | 64 |
| 5 | 62 |
| 6 | 66 |
| 7 | 66 |
| 8 | 65 |
| 9 | 69 |
| 10 | 66 |
| 11 | 73 |
| 12 | 69 |
| 13 | 69 |

| 14 | 61 |
|----|----|
| 15 | 64 |
| 16 | 61 |
| 17 | 71 |
| 18 | 67 |
| 19 | 73 |
| 20 | 66 |

**Transformation:**

Use the following transformations to calculate the median height in inches, a specified percentile and the first quartile.

- The first function uses a precise algorithm which can be slow to execute across large datasets.
- The second function uses an appropriate approximation algorithm, which is much faster to execute across large datasets.
    - These approximate functions can use an error boundary parameter, which is set to `0.4` (0.4%) across all functions.

**Median:** This transformation calculates the median value, which corresponds to the 50th percentile.

| Transformation Name | `New formula` |
|---|---|
| **Parameter: Formula type** | `Single row formula` |
| **Parameter: Formula** | `median(heightIn)` |
| **Parameter: New column name** | `'medianIn'` |

| Transformation Name | `New formula` |
|---|---|
| **Parameter: Formula type** | `Single row formula` |
| **Parameter: Formula** | `approximatemedian(heightIn, 0.4)` |
| **Parameter: New column name** | `'approxMedianIn'` |

**Percentile:** This transformation calculates the 68th percentile.

| Transformation Name | `New formula` |
|---|---|
| **Parameter: Formula type** | `Single row formula` |
| **Parameter: Formula** | `percentile(heightIn, 68, linear)` |
| **Parameter: New column name** | `'percentile68In'` |

| Transformation Name | `New formula` |
|---|---|
| **Parameter: Formula type** | `Single row formula` |
| **Parameter: Formula** | `approximatepercentile(heightIn, 68, 0.4)` |

| Parameter: New column name | 'approxPercentile68In' |
|---|---|

**Quartile:** This transformation calculates the first quartile, which corresponds to the 25th percentile.

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | quartile(heightIn, 1, linear) |
| Parameter: New column name | 'percentile25In' |

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | approximatequartile(heightIn, 1, 0.4) |
| Parameter: New column name | 'approxPercentile25In' |

**Results:**

| studentId | heightIn | approxPercentile25In | percentile25In | approxPercentile68In | percentile68In | approxMedianIn | |
|---|---|---|---|---|---|---|---|
| 1 | 64 | 64 | 64 | 67.1 | 66.92 | 66 | 6 |
| 2 | 65 | 64 | 64 | 67.1 | 66.92 | 66 | 6 |
| 3 | 63 | 64 | 64 | 67.1 | 66.92 | 66 | 6 |
| 4 | 64 | 64 | 64 | 67.1 | 66.92 | 66 | 6 |
| 5 | 62 | 64 | 64 | 67.1 | 66.92 | 66 | 6 |
| 6 | 66 | 64 | 64 | 67.1 | 66.92 | 66 | 6 |
| 7 | 66 | 64 | 64 | 67.1 | 66.92 | 66 | 6 |
| 8 | 65 | 64 | 64 | 67.1 | 66.92 | 66 | 6 |
| 9 | 69 | 64 | 64 | 67.1 | 66.92 | 66 | 6 |
| 10 | 66 | 64 | 64 | 67.1 | 66.92 | 66 | 6 |
| 11 | 73 | 64 | 64 | 67.1 | 66.92 | 66 | 6 |
| 12 | 69 | 64 | 64 | 67.1 | 66.92 | 66 | 6 |
| 13 | 69 | 64 | 64 | 67.1 | 66.92 | 66 | 6 |
| 14 | 61 | 64 | 64 | 67.1 | 66.92 | 66 | 6 |
| 15 | 64 | 64 | 64 | 67.1 | 66.92 | 66 | 6 |
| 16 | 61 | 64 | 64 | 67.1 | 66.92 | 66 | 6 |
| 17 | 71 | 64 | 64 | 67.1 | 66.92 | 66 | 6 |
| 18 | 67 | 64 | 64 | 67.1 | 66.92 | 66 | 6 |
| 19 | 73 | 64 | 64 | 67.1 | 66.92 | 66 | 6 |
| 20 | 66 | 64 | 64 | 67.1 | 66.92 | 66 | 6 |

# AVERAGE Function

Computes the average (mean) from all row values in a column or group. Input column can be of Integer or Decimal.

- If a row contains a missing or null value, it is not factored into the calculation. If the entire column contains no values, the function returns a null value.
- When used in a `pivot` transform, the function is computed for each instance of the value specified in the `group` parameter. See *Pivot Transform*.

For a version of this function computed over a rolling window of rows, see *ROLLINGAVERAGE Function*.

**Wrangle vs. SQL:** This function is part of Wrangle , a proprietary data transformation language. Wrangle is not SQL. For more information, see *Wrangle Language*.

## Basic Usage

```
average(myRating)
```

**Output:** Returns the average of the values in the `myRating` column.

## Syntax and Arguments

```
average(function_col_ref) [group:group_col_ref] [limit:limit_count]
```

| Argument | Required? | Data Type | Description |
|---|---|---|---|
| function_col_ref | Y | string | Name of column to which to apply the function |

For more information on the `group` and `limit` parameters, see *Pivot Transform*.

For more information on syntax standards, see *Language Documentation Syntax Notes*.

### function_col_ref

Name of the column the values of which you want to calculate the average. Column must contain Integer or Decimal values.

- Literal values are not supported as inputs.
- Multiple columns and wildcards are not supported.

**Usage Notes:**

| Required? | Data Type | Example Value |
|---|---|---|
| Yes | String (column reference) | `myValues` |

## Examples

> **Tip:** For additional examples, see *Common Tasks*.

**Example - Statistics on Test Scores**

This example illustrates how you can apply statistical functions to your dataset. Calculations include average (mean), max, min, standard deviation, and variance.

**Functions:**

| Item | Description |
|------|-------------|
| AVERAGE Function | Computes the average (mean) from all row values in a column or group. Input column can be of Integer or Decimal. |
| MIN Function | Computes the minimum value found in all row values in a column. Input column can be of Integer, Decimal or Datetime. |
| MAX Function | Computes the maximum value found in all row values in a column. Inputs can be Integer, Decimal, or Datetime. |
| VAR Function | Computes the variance among all values in a column. Input column can be of Integer or Decimal. If no numeric values are detected in the input column, the function returns 0. |
| STDEV Function | Computes the standard deviation across all column values of Integer or Decimal type. |
| NUMFORMAT Function | Formats a numeric set of values according to the specified number formatting. Source values can be a literal numeric value, a function returning a numeric value, or reference to a column containing an Integer or Decimal values. |
| MODE Function | Computes the mode (most frequent value) from all row values in a column, according to their grouping. Input column can be of Integer, Decimal, or Datetime type. |

**Source:**

Students took a test and recorded the following scores. You want to perform some statistical analysis on them:

| Student | Score |
|---------|-------|
| Anna | 84 |
| Ben | 71 |
| Caleb | 76 |
| Danielle | 87 |
| Evan | 85 |
| Faith | 92 |
| Gabe | 85 |
| Hannah | 99 |
| Ian | 73 |
| Jane | 68 |

**Transformation:**

You can use the following transformations to calculate the average (mean), minimum, and maximum scores:

| Transformation Name | New formula |
|---------------------|-------------|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | AVERAGE(Score) |

| Parameter: New column name | 'avgScore' |
|---|---|

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | MIN(Score) |
| Parameter: New column name | 'minScore' |

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | MAX(Score) |
| Parameter: New column name | 'maxScore' |

To apply statistical functions to your data, you can use the VAR and STDEV functions, which can be used as the basis for other statistical calculations.

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | VAR(Score) |
| Parameter: New column name | var_Score |

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | STDEV(Score) |
| Parameter: New column name | stdev_Score |

For each score, you can now calculate the variation of each one from the average, using the following:

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | ((Score - avg_Score) / stdev_Score) |
| Parameter: New column name | 'stDevs' |

Now, you want to apply grades based on a formula:

| Grade | standard deviations from avg (stDevs) |
|---|---|
| A | stDevs > 1 |
| B | stDevs > 0.5 |

| C | -1 <= stDevs <= 0.5 |
|---|---|
| D | stDevs < -1 |
| F | stDevs < -2 |

You can build the following transformation using the `IF` function to calculate grades.

| Transformation Name | `New formula` |
|---|---|
| Parameter: Formula type | `Single row formula` |
| Parameter: Formula | `IF((stDevs > 1),'A',IF((stDevs < -2),'F',IF((stDevs < -1),'D',IF`<br>`((stDevs > 0.5),'B','C'))))` |

To clean up the content, you might want to apply some formatting to the score columns. The following reformats the `stdev_Score` and `stDevs` columns to display two decimal places:

| Transformation Name | `Edit column with formula` |
|---|---|
| Parameter: Columns | `stdev_Score` |
| Parameter: Formula | `NUMFORMAT(stdev_Score, '##.00')` |

| Transformation Name | `Edit column with formula` |
|---|---|
| Parameter: Columns | `stDevs` |
| Parameter: Formula | `NUMFORMAT(stDevs, '##.00')` |

| Transformation Name | `New formula` |
|---|---|
| Parameter: Formula type | `Single row formula` |
| Parameter: Formula | `MODE(Score)` |
| Parameter: New column name | `'modeScore'` |

### Results:

| Student | Score | modeScore | avgScore | minScore | maxScore | var_Score | stdev_Score | stDevs | Grade |
|---|---|---|---|---|---|---|---|---|---|
| Anna | 84 | 85 | 82 | 68 | 99 | 87.000000000 00001 | 9.33 | 0.21 | C |
| Ben | 71 | 85 | 82 | 68 | 99 | 87.000000000 00001 | 9.33 | -1.18 | D |
| Caleb | 76 | 85 | 82 | 68 | 99 | 87.000000000 00001 | 9.33 | -0.64 | C |
| Danielle | 87 | 85 | 82 | 68 | 99 | 87.000000000 00001 | 9.33 | 0.54 | B |
| Evan | 85 | 85 | 82 | 68 | 99 | 87.000000000 00001 | 9.33 | 0.32 | C |
| Faith | 92 | 85 | 82 | 68 | 99 | 87.000000000 00001 | 9.33 | 1.07 | A |

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| Gabe | 85 | 85 | 82 | 68 | 99 | 87.000000000 00001 | 9.33 | 0.32 | C |
| Hannah | 99 | 85 | 82 | 68 | 99 | 87.000000000 00001 | 9.33 | 1.82 | A |
| Ian | 73 | 85 | 82 | 68 | 99 | 87.000000000 00001 | 9.33 | -0.96 | C |
| Jane | 68 | 85 | 82 | 68 | 99 | 87.000000000 00001 | 9.33 | -1.50 | D |

# AVERAGEIF Function

**Contents:**

- *Basic Usage*
- *Syntax and Arguments*
  - *col_ref*
  - *test_expression*
- *Examples*
  - *Example - Conditional Calculation Functions*

---

Generates the average value of rows in each group that meet a specific condition. Generated value is of Decimal type.

> **NOTE:** When added to a transformation, this function is applied to the current sample. If you change your sample or run the job, the computed values for this function are updated. Transformations that change the number of rows in subsequent recipe steps do not affect the values computed for this step.

To perform a simple averaging of rows without conditionals, use the `AVERAGE` function. See *AVERAGE Function*.

**Wrangle vs. SQL:** This function is part of Wrangle , a proprietary data transformation language. Wrangle is not SQL. For more information, see *Wrangle Language*.

## Basic Usage

```
averageif(fineValue, finesCount &gt;= 3)
```

**Output:** Generates a two-column table containing the unique values for `postal_code` and the average of the `fineValue` column for that `postal_code` value when the `finesCount` is greater than or equal to 3. The `limit` parameter defines the maximum number of output columns.

## Syntax and Arguments

```
averageif(col_ref, test_expression) [group:group_col_ref] [limit:limit_count]
```

| Argument | Required? | Data Type | Description |
|---|---|---|---|
| col_ref | Y | string | Reference to the column you wish to evaluate. |
| test_expression | Y | string | Expression that is evaluated. Must resolve to `true` or `false` |

For more information on syntax standards, see *Language Documentation Syntax Notes*.

For more information on the `group` and `limit` parameter, see *Pivot Transform*.

**col_ref**

Name of the column whose values you wish to use in the calculation. Column must be a numeric (Integer or Decimal) type.

**Usage Notes:**

| Required? | Data Type | Example Value |
|---|---|---|
| Yes | String that corresponds to the name of the column | `myValues` |

### test_expression

This parameter contains the expression to evaluate. This expression must resolve to a Boolean (`true` or `false`) value.

**Usage Notes:**

| Required? | Data Type | Example Value |
|---|---|---|
| Yes | String expression that evaluates to `true` or `false` | `(LastName == 'Mouse' && FirstName == 'Mickey')` |

## Examples

> **Tip:** For additional examples, see *Common Tasks*.

### Example - Conditional Calculation Functions

This example illustrates how to use the conditional calculation functions.

**Functions**:

| Item | Description |
|---|---|
| AVERAGEIF Function | Generates the average value of rows in each group that meet a specific condition. Generated value is of Decimal type. |
| MINIF Function | Generates the minimum value of rows in each group that meet a specific condition. Inputs can be Integer, Decimal, or Datetime. |
| MAXIF Function | Generates the maximum value of rows in each group that meet a specific condition. Inputs can be Integer, Decimal, or Datetime. |
| VARIF Function | Generates the variance of values by group in a column that meet a specific condition. |
| STDEVIF Function | Generates the standard deviation of values by group in a column that meet a specific condition. |

**Source:**

Here is some example weather data:

| date | city | rain | temp | wind |
|---|---|---|---|---|
| 1/23/17 | Valleyville | 0.00 | 12.8 | 6.7 |
| 1/23/17 | Center Town | 0.31 | 9.4 | 5.3 |
| 1/23/17 | Magic Mountain | 0.00 | 0.0 | 7.3 |
| 1/24/17 | Valleyville | 0.25 | 17.2 | 3.3 |

| 1/24/17 | Center Town | 0.54 | 1.1 | 7.6 |
|---------|-------------|------|------|-----|
| 1/24/17 | Magic Mountain | 0.32 | 5.0 | 8.8 |
| 1/25/17 | Valleyville | 0.02 | 3.3 | 6.8 |
| 1/25/17 | Center Town | 0.83 | 3.3 | 5.1 |
| 1/25/17 | Magic Mountain | 0.59 | -1.7 | 6.4 |
| 1/26/17 | Valleyville | 1.08 | 15.0 | 4.2 |
| 1/26/17 | Center Town | 0.96 | 6.1 | 7.6 |
| 1/26/17 | Magic Mountain | 0.77 | -3.9 | 3.0 |
| 1/27/17 | Valleyville | 1.00 | 7.2 | 2.8 |
| 1/27/17 | Center Town | 1.32 | 20.0 | 0.2 |
| 1/27/17 | Magic Mountain | 0.77 | 5.6 | 5.2 |
| 1/28/17 | Valleyville | 0.12 | -6.1 | 5.1 |
| 1/28/17 | Center Town | 0.14 | 5.0 | 4.9 |
| 1/28/17 | Magic Mountain | 1.50 | 1.1 | 0.4 |
| 1/29/17 | Valleyville | 0.36 | 13.3 | 7.3 |
| 1/29/17 | Center Town | 0.75 | 6.1 | 9.0 |
| 1/29/17 | Magic Mountain | 0.60 | 3.3 | 6.0 |

**Transformation:**

The following computes average temperature for rainy days by city:

| Transformation Name | New formula |
|---------------------|-------------|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | AVERAGEIF(temp, rain > 0) |
| Parameter: Group rows by | city |
| Parameter: New column name | 'avgTempWRain' |

The following computes maximum wind for sub-zero days by city:

| Transformation Name | New formula |
|---------------------|-------------|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | MAXIF(wind,temp < 0) |
| Parameter: Group rows by | city |
| Parameter: New column name | 'maxWindSubZero' |

This step calculates the minimum temp when the wind is less than 5 mph by city:

| Transformation Name | New formula |
|---------------------|-------------|

| Parameter: Formula type | Single row formula |
|---|---|
| Parameter: Formula | MINIF(temp,wind<5) |
| Parameter: Group rows by | city |
| Parameter: New column name | 'minTempWind5' |

This step computes the variance in temperature for rainy days by city:

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | VARIF(temp,rain >0) |
| Parameter: Group rows by | city |
| Parameter: New column name | 'varTempWRain' |

The following computes the standard deviation in rainfall for Center Town:

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | STDEVIF(rain,city=='Center Town') |
| Parameter: Group rows by | city |
| Parameter: New column name | 'stDevRainCT' |

You can use the following transforms to format the generated output. Note the `$col` placeholder value for the multi-column transforms:

| Transformation Name | Edit column with formula |
|---|---|
| Parameter: Columns | stDevRainCenterTown,maxWindSubZero |
| Parameter: Formula | numformat($col,'##.##') |

Since the following rely on data that has only one significant digit, you should format them differently:

| Transformation Name | Edit column with formula |
|---|---|
| Parameter: Columns | varTempWRain,avgTempWRain,minTempWind5 |
| Parameter: Formula | numformat($col,'##.#') |

**Results:**

| date | city | rain | temp | wind | avgTempWRain | maxWindSubZero | minTempWind5 | varTempWRain | stDevRain |
|---|---|---|---|---|---|---|---|---|---|
| 1/23/17 | Valley ville | 0.00 | 12.8 | 6.7 | 8.3 | 5.1 | 7.2 | 63.8 | 0.37 |

| 1/23/17 | Center Town | 0.31 | 9.4 | 5.3 | 7.3 | | 5 | 32.6 | 0.37 |
|---|---|---|---|---|---|---|---|---|---|
| 1/23/17 | Magic Mountain | 0.00 | 0.0 | 7.3 | 1.6 | 6.43 | -3.9 | 12 | 0.37 |
| 1/24/17 | Valleyville | 0.25 | 17.2 | 3.3 | 8.3 | 5.1 | 7.2 | 63.8 | 0.37 |
| 1/24/17 | Center Town | 0.54 | 1.1 | 7.6 | 7.3 | | 5 | 32.6 | 0.37 |
| 1/24/17 | Magic Mountain | 0.32 | 5.0 | 8.8 | 1.6 | 6.43 | -3.9 | 12 | 0.37 |
| 1/25/17 | Valleyville | 0.02 | 3.3 | 6.8 | 8.3 | 5.1 | 7.2 | 63.8 | 0.37 |
| 1/25/17 | Center Town | 0.83 | 3.3 | 5.1 | 7.3 | | 5 | 32.6 | 0.37 |
| 1/25/17 | Magic Mountain | 0.59 | -1.7 | 6.4 | 1.6 | 6.43 | -3.9 | 12 | 0.37 |
| 1/26/17 | Valleyville | 1.08 | 15.0 | 4.2 | 8.3 | 5.1 | 7.2 | 63.8 | 0.37 |
| 1/26/17 | Center Town | 0.96 | 6.1 | 7.6 | 7.3 | | 5 | 32.6 | 0.37 |
| 1/26/17 | Magic Mountain | 0.77 | -3.9 | 3.0 | 1.6 | 6.43 | -3.9 | 12 | 0.37 |
| 1/27/17 | Valleyville | 1.00 | 7.2 | 2.8 | 8.3 | 5.1 | 7.2 | 63.8 | 0.37 |
| 1/27/17 | Center Town | 1.32 | 20.0 | 0.2 | 7.3 | | 5 | 32.6 | 0.37 |
| 1/27/17 | Magic Mountain | 0.77 | 5.6 | 5.2 | 1.6 | 6.43 | -3.9 | 12 | 0.37 |
| 1/28/17 | Valleyville | 0.12 | -6.1 | 5.1 | 8.3 | 5.1 | 7.2 | 63.8 | 0.37 |
| 1/28/17 | Center Town | 0.14 | 5.0 | 4.9 | 7.3 | | 5 | 32.6 | 0.37 |
| 1/28/17 | Magic Mountain | 1.50 | 1.1 | 0.4 | 1.6 | 6.43 | -3.9 | 12 | 0.37 |
| 1/29/17 | Valleyville | 0.36 | 13.3 | 7.3 | 8.3 | 5.1 | 7.2 | 63.8 | 0.37 |
| 1/29/17 | Center Town | 0.75 | 6.1 | 9.0 | 7.3 | | 5 | 32.6 | 0.37 |
| 1/29/17 | Magic Mountain | 0.60 | 3.3 | 6.0 | 1.6 | 6.43 | -3.9 | 12 | 0.37 |

# CORREL Function

Computes the correlation coefficient between two columns. Source values can be of Integer or Decimal type.

The **correlation coefficient** measures the relationship between two sets of values. You can use it as a measurement for how changes in one value affect changes in the other.

- Values range between -1 (negative correlation) and +1 (positive correlation).
    - Negative correlation means that the second number tends to decrease when the first number increases.
    - Positive correlation means that the second number tends to increase when the first number increases.
    - A correlation coefficient that is close to 0 indicates a weak or non-existent correlation.

**Relevant terms:**

| Term | Description |
|------|-------------|
| Population | Population statistical functions are computed from all possible values. See *https://en.wikipedia.org/wiki/Statistical_population*. |
| Sample | Sample-based statistical functions are computed from a subset or sample of all values. See *https://en.wikipedia.org/wiki/Sampling_(statistics)*. <br><br> These function names include SAMP in their name. <br><br> **NOTE:** Statistical sampling has no relationship to the samples taken within the product. When statistical functions are computed during job execution, they are applied across the entire dataset. Sample method calculations are computed at that time. |

**Wrangle vs. SQL:** This function is part of Wrangle , a proprietary data transformation language. Wrangle is not SQL. For more information, see *Wrangle Language*.

## Basic Usage

```
correl(initialInvestment,ROI)
```

**Output:** Returns the correlation coefficient between the values in the `initialInvestment` column and the `ROI` column.

## Syntax and Arguments

```
correl(function_col_ref1,<span>function_col_ref2</span>) [group:group_col_ref] [limit:
limit_count]
```

| Argument | Required? | Data Type | Description |
|----------|-----------|-----------|-------------|
| function_col_ref1 | Y | string | Name of column that is the first input to the function |
| function_col_ref2 | Y | string | Name of column that is the second input to the function |

For more information on the `group` and `limit` parameters, see *Pivot Transform*.

For more information on syntax standards, see *Language Documentation Syntax Notes*.

**function_col_ref1, function_col_ref2**

Name of the column the values of which you want to calculate the correlation. Column must contain Integer or Decimal values.

- Literal values are not supported as inputs.
- Multiple columns and wildcards are not supported.

**Usage Notes:**

| Required? | Data Type | Example Value |
|-----------|-----------|---------------|
| Yes | String (column reference) | `myInputs` |

## Examples

> **Tip:** For additional examples, see *Common Tasks*.

This example illustrates statistical functions that can be applied across two columns of values.

**Functions**:

| Item | Description |
|------|-------------|
| CORREL Function | Computes the correlation coefficient between two columns. Source values can be of Integer or Decimal type. |
| COVAR Function | Computes the covariance between two columns using the population method. Source values can be of Integer or Decimal type. |
| COVARSAMP Function | Computes the covariance between two columns using the sample method. Source values can be of Integer or Decimal type. |
| ROUND Function | Rounds input value to the nearest integer. Input can be an Integer, a Decimal, a column reference, or an expression. Optional second argument can be used to specify the number of digits to which to round. |

**Source:**

The following table contains height in inches and weight in pounds for a set of students.

| Student | heightIn | weightLbs |
|---------|----------|-----------|
| 1 | 70 | 134 |
| 2 | 67 | 135 |
| 3 | 67 | 147 |
| 4 | 67 | 160 |
| 5 | 72 | 136 |
| 6 | 73 | 146 |
| 7 | 71 | 135 |
| 8 | 63 | 145 |
| 9 | 67 | 138 |

| | | |
|---|---|---|
| 10 | 66 | 138 |
| 11 | 71 | 161 |
| 12 | 70 | 131 |
| 13 | 74 | 131 |
| 14 | 67 | 157 |
| 15 | 73 | 161 |
| 16 | 70 | 133 |
| 17 | 63 | 132 |
| 18 | 64 | 153 |
| 19 | 64 | 156 |
| 20 | 72 | 154 |

**Transformation:**

You can use the following transformations to calculate the correlation co-efficient, the covariance, and the sampling method covariance between the two data columns:

| | |
|---|---|
| **Transformation Name** | New formula |
| **Parameter: Formula type** | Single row formula |
| **Parameter: Formula** | round(correl(heightIn, weightLbs), 3) |
| **Parameter: New column name** | 'corrHeightAndWeight' |

| | |
|---|---|
| **Transformation Name** | New formula |
| **Parameter: Formula type** | Single row formula |
| **Parameter: Formula** | round(covar(heightIn, weightLbs), 3) |
| **Parameter: New column name** | 'covarHeightAndWeight' |

| | |
|---|---|
| **Transformation Name** | New formula |
| **Parameter: Formula type** | Single row formula |
| **Parameter: Formula** | round(covarsamp(heightIn, weightLbs), 3) |
| **Parameter: New column name** | 'covarHeightAndWeight-Sample' |

**Results:**

| Student | heightIn | weightLbs | covarHeightAndWeight-Sample | covarHeightAndWeight | corrHeightAndWeight |
|---|---|---|---|---|---|
| 1 | 70 | 134 | -2.876 | -2.732 | -0.074 |
| 2 | 67 | 135 | -2.876 | -2.732 | -0.074 |
| 3 | 67 | 147 | -2.876 | -2.732 | -0.074 |

| 4 | 67 | 160 | -2.876 | -2.732 | -0.074 |
|---|----|-----|--------|--------|--------|
| 5 | 72 | 136 | -2.876 | -2.732 | -0.074 |
| 6 | 73 | 146 | -2.876 | -2.732 | -0.074 |
| 7 | 71 | 135 | -2.876 | -2.732 | -0.074 |
| 8 | 63 | 145 | -2.876 | -2.732 | -0.074 |
| 9 | 67 | 138 | -2.876 | -2.732 | -0.074 |
| 10 | 66 | 138 | -2.876 | -2.732 | -0.074 |
| 11 | 71 | 161 | -2.876 | -2.732 | -0.074 |
| 12 | 70 | 131 | -2.876 | -2.732 | -0.074 |
| 13 | 74 | 131 | -2.876 | -2.732 | -0.074 |
| 14 | 67 | 157 | -2.876 | -2.732 | -0.074 |
| 15 | 73 | 161 | -2.876 | -2.732 | -0.074 |
| 16 | 70 | 133 | -2.876 | -2.732 | -0.074 |
| 17 | 63 | 132 | -2.876 | -2.732 | -0.074 |
| 18 | 64 | 153 | -2.876 | -2.732 | -0.074 |
| 19 | 64 | 156 | -2.876 | -2.732 | -0.074 |
| 20 | 72 | 154 | -2.876 | -2.732 | -0.074 |

# COUNTA Function

Generates the count of non-null rows in a specified column, optionally counted by group. Generated value is of Integer type.

> **NOTE:** Empty string values are counted. Null values are not counted.

> **NOTE:** When added to a transformation, this function calculates the number of values in the specified column, as displayed in the current sample. Counts are not applied to the entire dataset until you run the job. If you change your sample or run the job, the computed values for this function are updated. Transformations that change the number of rows in subsequent recipe steps do not affect the value for the already computed instance of `COUNTA`.

For a version of this function computed over a rolling window of rows, see *ROLLINGCOUNTA Function*.

**Wrangle vs. SQL:** This function is part of Wrangle , a proprietary data transformation language. Wrangle is not SQL. For more information, see *Wrangle Language*.

## Basic Usage

```
counta(name)
```

**Output:** Returns the count of non-empty values in the `name` column.

## Syntax and Arguments

```
counta(function_col_ref) [group:group_col_ref] [limit:limit_count]
```

| Argument | Required? | Data Type | Description |
|---|---|---|---|
| function_col_ref | Y | string | Name of column to which to apply the function |

For more information on the `group` parameter, see *Pivot Transform*.

**function_col_ref**

Name of the column from which to count values based on the grouping.

- Literal values are not supported as inputs.
- Multiple columns and wildcards are not supported.

**Usage Notes:**

| Required? | Data Type | Example Value |
|---|---|---|
| Yes | String (column reference) | `myValues` |

# Examples

> **Tip:** For additional examples, see *Common Tasks*.

### Example - Simple row count

This example demonstrates how to count the number of values and non-null values within a group.

**Functions**:

| Item | Description |
|------|-------------|
| COUNTA Function | Generates the count of non-null rows in a specified column, optionally counted by group. Generated value is of Integer type. |
| COUNTDISTINCT Function | Generates the count of distinct values in a specified column, optionally counted by group. Generated value is of Integer type. |

### Source:

In the following example, the seventh row is an empty string, and the eighth row is a null value.

| rowId | Val |
|-------|-----|
| r001 | val1 |
| r002 | val1 |
| r003 | val1 |
| r004 | val2 |
| r005 | val2 |
| r006 | val3 |
| r007 | (empty) |
| r008 | (null) |

### Transformation:

Apply a `COUNTA` function on the source column:

| | |
|------|------|
| **Transformation Name** | New formula |
| **Parameter: Formula type** | Single row formula |
| **Parameter: Formula** | COUNTA(Val) |
| **Parameter: New column name** | 'fctnCounta' |

Apply a `COUNTDISTINCT` function on the source:

| | |
|------|------|
| **Transformation Name** | New formula |
| **Parameter: Formula type** | Single row formula |
| **Parameter: Formula** | COUNTDISTINCT(Val) |

| Parameter: New column name | 'fctnCountdistinct' |
|---|---|

**Results:**

Below, both functions count the number of values in the column, with `COUNTDISTINCT` counting distinct values only. The empty value for `r007` is counted by both functions.

| rowId | Val | fctnCountdistinct | fctnCounta |
|---|---|---|---|
| r001 | val1 | 4 | 7 |
| r002 | val1 | 4 | 7 |
| r003 | val1 | 4 | 7 |
| r004 | val2 | 4 | 7 |
| r005 | val2 | 4 | 7 |
| r006 | val3 | 4 | 7 |
| r007 | (empty) | 4 | 7 |
| r008 | (null) | 4 | 7 |

# COUNTAIF Function

**Contents:**

Generates the count of non-null values for rows in each group that meet a specific condition.

> **NOTE:** When added to a transformation, this function is applied to the current sample. If you change your sample or run the job, the computed values for this function are updated. Transformations that change the number of rows in subsequent recipe steps do not affect the values computed for this step.

To perform a simple counting of non-nulls without conditionals, use the COUNTA function. See *COUNTA Function*.

**Wrangle vs. SQL:** This function is part of Wrangle , a proprietary data transformation language. Wrangle is not SQL. For more information, see *Wrangle Language*.

## Basic Usage

```
countaif(entries, entryValidation == &apos;Ok&apos;)
```

**Output:** Returns the count of non-null values in the `entries` column when the `entryValidation` value is `'Ok'`
.

## Syntax and Arguments

```
countaif(col_ref, test_expression) [group:group_col_ref] [limit:limit_count]
```

| Argument | Required? | Data Type | Description |
|---|---|---|---|
| col_ref | Y | string | Reference to the column you wish to evaluate. |
| test_expression | Y | string | Expression that is evaluated. Must resolve to `true` or `false` |

For more information on syntax standards, see *Language Documentation Syntax Notes*.

For more information on the `group` parameter, see *Pivot Transform*.

**col_ref**

Name of the column whose values you wish to use in the calculation. Column must be a numeric (Integer or Decimal) type.

**Usage Notes:**

| Required? | Data Type | Example Value |
|---|---|---|
| Yes | String that corresponds to the name of the column | `myValues` |

### test_expression

This parameter contains the expression to evaluate. This expression must resolve to a Boolean (`true` or `false`) value.

**Usage Notes:**

| Required? | Data Type | Example Value |
|---|---|---|
| Yes | String expression that evaluates to `true` or `false` | `(LastName == 'Mouse' && FirstName == 'Mickey')` |

## Examples

> **Tip:** For additional examples, see *Common Tasks*.

### Example - COUNTIF Functions

This example demonstrates how to count the number of values within a group, based on a specified conditional test.

**Functions**:

| Item | Description |
|---|---|
| COUNTIF Function | Generates the count of rows in each group that meet a specific condition. Generated value is of Integer type. |
| COUNTAIF Function | Generates the count of non-null values for rows in each group that meet a specific condition. |

**Source:**

The following data identifies sales figures by salespeople for a week:

| EmployeeId | Date | Sales |
|---|---|---|
| S001 | 1/23/17 | 25 |
| S002 | 1/23/17 | 40 |
| S003 | 1/23/17 | 48 |
| S001 | 1/24/17 | 81 |
| S002 | 1/24/17 | 11 |
| S003 | 1/24/17 | 25 |
| S001 | 1/25/17 | 9 |
| S002 | 1/25/17 | 40 |

| | | |
|---|---|---|
| S003 | 1/25/17 | |
| S001 | 1/26/17 | 77 |
| S002 | 1/26/17 | 83 |
| S003 | 1/26/17 | |
| S001 | 1/27/17 | 17 |
| S002 | 1/27/17 | 71 |
| S003 | 1/27/17 | 29 |
| S001 | 1/28/17 | |
| S002 | 1/28/17 | |
| S003 | 1/28/17 | 14 |
| S001 | 1/29/17 | 2 |
| S002 | 1/29/17 | 7 |
| S003 | 1/29/17 | 99 |

**Transformation:**

You are interested in the count of dates during the week when each salesperson sold less than 50 units, not factoring the weekend. First, you try the following:

| | |
|---|---|
| **Transformation Name** | `Pivot columns` |
| **Parameter: Row labels** | `EmployeeId` |
| **Parameter: Values** | `COUNTIF(Sales < 50)` |
| **Parameter: Max columns to create** | `1` |

You notice, however, that the blank values, when employees were sick or had vacation, are being counted. Additionally, this step does not filter out the weekend. You must identify the weekend days using the `WEEKDAY` function:

| | |
|---|---|
| **Transformation Name** | `New formula` |
| **Parameter: Formula type** | `Single row formula` |
| **Parameter: Formula** | `WEEKDAY(Date)` |
| **Parameter: New column name** | `'DayOfWeek'` |

If `DayOfWeek > 5`, then it is a weekend date. For further precision, you can use the `COUNTAIF` function to remove the nulls:

| | |
|---|---|
| **Transformation Name** | `Pivot columns` |
| **Parameter: Row labels** | `EmployeeId` |
| **Parameter: Values** | `COUNTAIF(Sales, DayOfWeek<6)` |

| Parameter: Max columns to create | 1 |
|---|---|

The above counts the non-null values in `Sales` when the day of the week is not a weekend day, as grouped by individual employee.

**Results:**

| EmployeeId | countaif_Sales |
|---|---|
| S001 | 5 |
| S002 | 4 |
| S003 | 4 |

# COUNTDISTINCT Function

Generates the count of distinct values in a specified column, optionally counted by group. Generated value is of Integer type.

> **NOTE:** Empty string values are counted. Null values are not counted.

> **NOTE:** When added to a transformation, the function calculates the number of distinct values in the specified column, as displayed in the current sample. Counts are not applied to the entire dataset until you run the job. If you change your sample or run the job, the computed values for this function are updated. Transformations that change the number of rows in subsequent recipe steps do not affect the value for the already computed instance of COUNTDISTINCT.

**Wrangle vs. SQL:** This function is part of Wrangle , a proprietary data transformation language. Wrangle is not SQL. For more information, see *Wrangle Language*.

## Basic Usage

```
countdistinct(name)
```

**Output:** Returns the count of distinct values in the `name` column.

## Syntax and Arguments

```
countdistinct(function_col_ref) [group:group_col_ref] [limit:limit_count]
```

| Argument | Required? | Data Type | Description |
|---|---|---|---|
| function_col_ref | Y | string | Name of column to which to apply the function |

For more information on the `group` and `limit` parameter, see *Pivot Transform*.

**function_col_ref**

Name of the column from which to count values based on the grouping.

- Literal values are not supported as inputs.
- Multiple columns and wildcards are not supported.

**Usage Notes:**

| Required? | Data Type | Example Value |
|---|---|---|
| Yes | String (column reference) | `myValues` |

## Examples

> **Tip:** For additional examples, see *Common Tasks*.

## Example - Simple row count

This example demonstrates how to count the number of values and non-null values within a group.

**Functions**:

| Item | Description |
|------|-------------|
| COUNTA Function | Generates the count of non-null rows in a specified column, optionally counted by group. Generated value is of Integer type. |
| COUNTDISTINCT Function | Generates the count of distinct values in a specified column, optionally counted by group. Generated value is of Integer type. |

### Source:

In the following example, the seventh row is an empty string, and the eighth row is a null value.

| rowId | Val |
|-------|-----|
| r001 | val1 |
| r002 | val1 |
| r003 | val1 |
| r004 | val2 |
| r005 | val2 |
| r006 | val3 |
| r007 | (empty) |
| r008 | (null) |

### Transformation:

Apply a COUNTA function on the source column:

| Transformation Name | New formula |
|---------------------|-------------|
| **Parameter: Formula type** | Single row formula |
| **Parameter: Formula** | COUNTA(Val) |
| **Parameter: New column name** | 'fctnCounta' |

Apply a COUNTDISTINCT function on the source:

| Transformation Name | New formula |
|---------------------|-------------|
| **Parameter: Formula type** | Single row formula |
| **Parameter: Formula** | COUNTDISTINCT(Val) |
| **Parameter: New column name** | 'fctnCountdistinct' |

### Results:

Below, both functions count the number of values in the column, with `COUNTDISTINCT` counting distinct values only. The empty value for `r007` is counted by both functions.

| rowId | Val | fctnCountdistinct | fctnCounta |
|-------|---------|-------------------|------------|
| r001 | val1 | 4 | 7 |
| r002 | val1 | 4 | 7 |
| r003 | val1 | 4 | 7 |
| r004 | val2 | 4 | 7 |
| r005 | val2 | 4 | 7 |
| r006 | val3 | 4 | 7 |
| r007 | (empty) | 4 | 7 |
| r008 | (null) | 4 | 7 |

# COUNTDISTINCTIF Function

**Contents:**

Generates the count of distinct non-null values for rows in each group that meet a specific condition.

> **NOTE:** When added to a transformation, this function is applied to the current sample. If you change your sample or run the job, the computed values for this function are updated. Transformations that change the number of rows in subsequent recipe steps do not affect the values computed for this step.

To perform a simple counting of distinct non-nulls without conditionals, use the `COUNTDISTINCT` function. See *COUNTDISTINCT Function.*

**Wrangle vs. SQL:** This function is part of Wrangle , a proprietary data transformation language. Wrangle is not SQL. For more information, see *Wrangle Language.*

## Basic Usage

```
countdistinctif(entries, entryValidation == 'Ok')
```

**Output:** Generates a two-column table containing the unique values for `City` and the count of distinct non-null values in the `entries` column for that `City` value when the `entryValidation` value is `'Ok'`. The `limit` para meter defines the maximum number of output columns.

## Syntax and Arguments

```
countdistinctif(col_ref, test_expression) [group:group_col_ref] [limit:limit_count]
```

| Argument | Required? | Data Type | Description |
|---|---|---|---|
| col_ref | Y | string | Reference to the column you wish to evaluate. |
| test_expression | Y | string | Expression that is evaluated. Must resolve to `true` or `false` |

For more information on syntax standards, see *Language Documentation Syntax Notes.*

For more information on the `group` and `limit` parameter, see *Pivot Transform.*

**col_ref**

Name of the column whose values you wish to use in the calculation. Column must be a numeric (Integer or Decimal) type.

**Usage Notes:**

| Required? | Data Type | Example Value |
|---|---|---|
| Yes | String that corresponds to the name of the column | `myValues` |

**test_expression**

This parameter contains the expression to evaluate. This expression must resolve to a Boolean (`true` or `false`) value.

**Usage Notes:**

| Required? | Data Type | Example Value |
|---|---|---|
| Yes | String expression that evaluates to `true` or `false` | `(LastName == 'Mouse' && FirstName == 'Mickey')` |

## Examples

> **Tip:** For additional examples, see *Common Tasks*.

**Example - Summarize Voter Registrations**

This example illustrates how you can use conditional calculation functions.

**Functions**:

| Item | Description |
|---|---|
| SUMIF Function | Generates the sum of rows in each group that meet a specific condition. |
| COUNTDISTINCTIF Function | Generates the count of distinct non-null values for rows in each group that meet a specific condition. |

**Source:**

Here is some example polling data across 16 precincts in 8 cities across 4 counties, where registrations have been invalidated at the polling station, preventing voters from voting. Precincts where this issue has occurred previously have been added to a watch list (`precinctWatchList`).

| totalReg | invalidReg | precinctWatchList | precinctId | cityId | countyId |
|---|---|---|---|---|---|
| 731 | 24 | y | 1 | 1 | 1 |
| 743 | 29 | y | 2 | 1 | 1 |
| 874 | 0 | | 3 | 2 | 1 |
| 983 | 0 | | 4 | 2 | 1 |
| 622 | 29 | | 5 | 3 | 2 |
| 693 | 0 | | 6 | 3 | 2 |
| 775 | 37 | y | 7 | 4 | 2 |
| 1025 | 49 | y | 8 | 4 | 2 |

| 787 | 13 |   | 9  | 5 | 3 |
|-----|----|---|----|---|---|
| 342 | 0  |   | 10 | 5 | 3 |
| 342 | 39 | y | 11 | 6 | 3 |
| 387 | 28 | y | 12 | 6 | 3 |
| 582 | 59 |   | 13 | 7 | 4 |
| 244 | 0  |   | 14 | 7 | 4 |
| 940 | 6  | y | 15 | 8 | 4 |
| 901 | 4  | y | 16 | 8 | 4 |

**Transformation:**

First, you want to sum up the invalid registrations (`invalidReg`) for precincts that are already on the watchlist (`precinctWatchList = y`). These sums are grouped by city, which can span multiple precincts:

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | SUMIF(invalidReg, precinctWatchList == "y") |
| Parameter: Group rows by | cityId |
| Parameter: New column name | 'invalidRegbyCityId' |

The `invalidRegbyCityId` column contains invalid registrations across the entire city.

Now, at the county level, you want to identify the number of precincts that were on the watch list and were part of a city-wide registration problem.

In the following step, the number of cities in each count are counted where invalid registrations within a city is greater than 60.

- This step creates a pivot aggregation.

| Transformation Name | Pivot columns |
|---|---|
| Parameter: Row labels | countyId |
| Parameter: Values | COUNTDISTINCTIF(precinctId, invalidRegbyCityId > 60) |
| Parameter: Max number of columns to create | 1 |

**Results:**

| countyId | countdistinctif_precinctId |
|---|---|
| 1 | 0 |
| 2 | 2 |
| 3 | 2 |
| 4 | 0 |

The voting officials in counties 2 and 3 should investigate their precinct registration issues.

# COUNT Function

Generates the count of rows in the dataset. Generated value is of Integer type.

> **NOTE:** When added to a transformation, this function calculates the number of rows displayed in the current sample and are not applied to the entire dataset until you run the job. If you change your sample or run the job, the computed values for this function are updated. Transformations that change the number of rows in subsequent recipe steps do not affect the value for the already computed instance of `COUNT`.

**Wrangle vs. SQL:** This function is part of Wrangle , a proprietary data transformation language. Wrangle is not SQL. For more information, see *Wrangle Language*.

## Basic Usage

```
count()
```

**Output:** Returns the count of records for the dataset.

## Syntax and Arguments

There are no arguments for this function.

You can use the `group` and `limit` parameters to specify the scope of the count. For more information on the `group` and `limit` parameters, see *Pivot Transform*.

## Examples

> **Tip:** For additional examples, see *Common Tasks*.

**Example - Simple row count**

**Source:**

| RowNum |
|--------|
| Row 1 |
| Row 2 |
| Row 3 |
| Row 4 |
| Row 5 |

**Transformation:**

| Transformation Name | New formula |
|---------------------|-------------|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | count() |

| | |
|---|---|
| **Parameter: New column name** | `'row_count'` |

| source | row_count |
|---|---|
| Row 1 | 5 |
| Row 2 | 5 |
| Row 3 | 5 |
| Row 4 | 5 |
| Row 5 | 5 |

# COUNTIF Function

Generates the count of rows in each group that meet a specific condition. Generated value is of Integer type.

> **NOTE:** When added to a transformation, this function is applied to the current sample. If you change your sample or run the job, the computed values for this function are updated. Transformations that change the number of rows in subsequent recipe steps do not affect the values computed for this step.

To perform a simple count of rows without conditionals, use the `COUNT` function. See *COUNT Function*.

**Wrangle vs. SQL:** This function is part of Wrangle , a proprietary data transformation language. Wrangle is not SQL. For more information, see *Wrangle Language*.

## Basic Usage

```
countif(failed_deliveries &gt;= 10)
```

**Output:** Returns the count of records in which the value of the `failed_deliveries` column is greater than or equal to 10.

## Syntax and Arguments

```
countif(test_expression) [group:group_col_ref] [limit:limit_count]
```

| Argument | Required? | Data Type | Description |
|---|---|---|---|
| test_expression | Y | string | Expression that is evaluated. Must resolve to `true` or `false` |

For more information on syntax standards, see *Language Documentation Syntax Notes*.

For more information on the `group` and `limit` parameters, see *Pivot Transform*.

**test_expression**

This parameter contains the expression to evaluate. This expression must resolve to a Boolean (`true` or `false`) value.

**Usage Notes:**

| Required? | Data Type | Example Value |
|---|---|---|
| Yes | String expression that evaluates to `true` or `false` | `(LastName == 'Mouse' && FirstName == 'Mickey')` |

## Examples

> **Tip:** For additional examples, see *Common Tasks*.

### Example - COUNTIF Functions

This example demonstrates how to count the number of values within a group, based on a specified conditional test.

**Functions**:

| Item | Description |
|---|---|
| COUNTIF Function | Generates the count of rows in each group that meet a specific condition. Generated value is of Integer type. |
| COUNTAIF Function | Generates the count of non-null values for rows in each group that meet a specific condition. |

### Source:

The following data identifies sales figures by salespeople for a week:

| EmployeeId | Date | Sales |
|---|---|---|
| S001 | 1/23/17 | 25 |
| S002 | 1/23/17 | 40 |
| S003 | 1/23/17 | 48 |
| S001 | 1/24/17 | 81 |
| S002 | 1/24/17 | 11 |
| S003 | 1/24/17 | 25 |
| S001 | 1/25/17 | 9 |
| S002 | 1/25/17 | 40 |
| S003 | 1/25/17 | |
| S001 | 1/26/17 | 77 |
| S002 | 1/26/17 | 83 |
| S003 | 1/26/17 | |
| S001 | 1/27/17 | 17 |
| S002 | 1/27/17 | 71 |
| S003 | 1/27/17 | 29 |
| S001 | 1/28/17 | |
| S002 | 1/28/17 | |
| S003 | 1/28/17 | 14 |
| S001 | 1/29/17 | 2 |
| S002 | 1/29/17 | 7 |
| S003 | 1/29/17 | 99 |

### Transformation:

You are interested in the count of dates during the week when each salesperson sold less than 50 units, not factoring the weekend. First, you try the following:

| Transformation Name | Pivot columns |
|---|---|
| **Parameter: Row labels** | `EmployeeId` |
| **Parameter: Values** | `COUNTIF(Sales < 50)` |
| **Parameter: Max columns to create** | `1` |

You notice, however, that the blank values, when employees were sick or had vacation, are being counted. Additionally, this step does not filter out the weekend. You must identify the weekend days using the `WEEKDAY` function:

| Transformation Name | New formula |
|---|---|
| **Parameter: Formula type** | `Single row formula` |
| **Parameter: Formula** | `WEEKDAY(Date)` |
| **Parameter: New column name** | `'DayOfWeek'` |

If `DayOfWeek > 5`, then it is a weekend date. For further precision, you can use the `COUNTAIF` function to remove the nulls:

| Transformation Name | Pivot columns |
|---|---|
| **Parameter: Row labels** | `EmployeeId` |
| **Parameter: Values** | `COUNTAIF(Sales, DayOfWeek<6)` |
| **Parameter: Max columns to create** | `1` |

The above counts the non-null values in `Sales` when the day of the week is not a weekend day, as grouped by individual employee.

**Results:**

| EmployeeId | countaif_Sales |
|---|---|
| S001 | 5 |
| S002 | 4 |
| S003 | 4 |

# COVAR Function

Computes the covariance between two columns using the population method. Source values can be of Integer or Decimal type.

**Covariance** measures the joint variation between two sets of values. The sign of the covariance tends to show the linear relationship between the two datasets; positive covariance indicates that the numbers tend to increase with each other.

- The magnitude of the covariance is difficult to interpret, as it varies with the size of the source values.
- The normalized version of covariance is the correlation coefficient, in which covariance is normalized between -1 and 1. For more information, see *CORREL Function*.

**Relevant terms:**

| Term | Description |
|------|-------------|
| Population | Population statistical functions are computed from all possible values. See *https://en.wikipedia.org/wiki/Statistical_population*. |
| Sample | Sample-based statistical functions are computed from a subset or sample of all values. See *https://en.wikipedia.org/wiki/Sampling_(statistics)*. <br><br> These function names include SAMP in their name. <br><br> **NOTE:** Statistical sampling has no relationship to the samples taken within the product. When statistical functions are computed during job execution, they are applied across the entire dataset. Sample method calculations are computed at that time. |

- This function is calculated across the entire population.
- For more information on a sampled version of this function, see *COVARSAMP Function*.

**Wrangle vs. SQL:** This function is part of Wrangle , a proprietary data transformation language. Wrangle is not SQL. For more information, see *Wrangle Language*.

## Basic Usage

```
covar(squareFootage,purchasePrice)
```

**Output:** Returns the covariance between the values in the `squareFootage` column and the `purchasePrice` column.

## Syntax and Arguments

```
covar(function_col_ref1,<span>function_col_ref2</span>) [group:group_col_ref] [limit:
limit_count]
```

| Argument | Required? | Data Type | Description |
|----------|-----------|-----------|-------------|
| function_col_ref1 | Y | string | Name of column that is the first input to the function |
| function_col_ref2 | Y | string | Name of column that is the second input to the function |

For more information on the `group` and `limit` parameters, see *Pivot Transform*.

For more information on syntax standards, see *Language Documentation Syntax Notes*.

**function_col_ref1, function_col_ref2**

Name of the column the values of which you want to calculate the covariance. Column must contain Integer or Decimal values.

- Literal values are not supported as inputs.
- Multiple columns and wildcards are not supported.

**Usage Notes:**

| Required? | Data Type | Example Value |
|---|---|---|
| Yes | String (column reference) | `myInputs` |

## Examples

> **Tip:** For additional examples, see *Common Tasks*.

This example illustrates statistical functions that can be applied across two columns of values.

**Functions**:

| Item | Description |
|---|---|
| CORREL Function | Computes the correlation coefficient between two columns. Source values can be of Integer or Decimal type. |
| COVAR Function | Computes the covariance between two columns using the population method. Source values can be of Integer or Decimal type. |
| COVARSAMP Function | Computes the covariance between two columns using the sample method. Source values can be of Integer or Decimal type. |
| ROUND Function | Rounds input value to the nearest integer. Input can be an Integer, a Decimal, a column reference, or an expression. Optional second argument can be used to specify the number of digits to which to round. |

**Source:**

The following table contains height in inches and weight in pounds for a set of students.

| Student | heightIn | weightLbs |
|---|---|---|
| 1 | 70 | 134 |
| 2 | 67 | 135 |
| 3 | 67 | 147 |
| 4 | 67 | 160 |
| 5 | 72 | 136 |
| 6 | 73 | 146 |
| 7 | 71 | 135 |
| 8 | 63 | 145 |
| 9 | 67 | 138 |

| 10 | 66 | 138 |
| --- | --- | --- |
| 11 | 71 | 161 |
| 12 | 70 | 131 |
| 13 | 74 | 131 |
| 14 | 67 | 157 |
| 15 | 73 | 161 |
| 16 | 70 | 133 |
| 17 | 63 | 132 |
| 18 | 64 | 153 |
| 19 | 64 | 156 |
| 20 | 72 | 154 |

### Transformation:

You can use the following transformations to calculate the correlation co-efficient, the covariance, and the sampling method covariance between the two data columns:

| Transformation Name | New formula |
| --- | --- |
| Parameter: Formula type | Single row formula |
| Parameter: Formula | round(correl(heightIn, weightLbs), 3) |
| Parameter: New column name | 'corrHeightAndWeight' |

| Transformation Name | New formula |
| --- | --- |
| Parameter: Formula type | Single row formula |
| Parameter: Formula | round(covar(heightIn, weightLbs), 3) |
| Parameter: New column name | 'covarHeightAndWeight' |

| Transformation Name | New formula |
| --- | --- |
| Parameter: Formula type | Single row formula |
| Parameter: Formula | round(covarsamp(heightIn, weightLbs), 3) |
| Parameter: New column name | 'covarHeightAndWeight-Sample' |

### Results:

| Student | heightIn | weightLbs | covarHeightAndWeight-Sample | covarHeightAndWeight | corrHeightAndWeight |
| --- | --- | --- | --- | --- | --- |
| 1 | 70 | 134 | -2.876 | -2.732 | -0.074 |
| 2 | 67 | 135 | -2.876 | -2.732 | -0.074 |
| 3 | 67 | 147 | -2.876 | -2.732 | -0.074 |

| 4 | 67 | 160 | -2.876 | -2.732 | -0.074 |
|---|----|-----|--------|--------|--------|
| 5 | 72 | 136 | -2.876 | -2.732 | -0.074 |
| 6 | 73 | 146 | -2.876 | -2.732 | -0.074 |
| 7 | 71 | 135 | -2.876 | -2.732 | -0.074 |
| 8 | 63 | 145 | -2.876 | -2.732 | -0.074 |
| 9 | 67 | 138 | -2.876 | -2.732 | -0.074 |
| 10 | 66 | 138 | -2.876 | -2.732 | -0.074 |
| 11 | 71 | 161 | -2.876 | -2.732 | -0.074 |
| 12 | 70 | 131 | -2.876 | -2.732 | -0.074 |
| 13 | 74 | 131 | -2.876 | -2.732 | -0.074 |
| 14 | 67 | 157 | -2.876 | -2.732 | -0.074 |
| 15 | 73 | 161 | -2.876 | -2.732 | -0.074 |
| 16 | 70 | 133 | -2.876 | -2.732 | -0.074 |
| 17 | 63 | 132 | -2.876 | -2.732 | -0.074 |
| 18 | 64 | 153 | -2.876 | -2.732 | -0.074 |
| 19 | 64 | 156 | -2.876 | -2.732 | -0.074 |
| 20 | 72 | 154 | -2.876 | -2.732 | -0.074 |

# COVARSAMP Function

Computes the covariance between two columns using the sample method. Source values can be of Integer or Decimal type.

**Covariance** measures the joint variation between two sets of values. The sign of the covariance tends to show the linear relationship between the two datasets; positive covariance indicates that the numbers tend to increase with each other.

- The magnitude of the covariance is difficult to interpret, as it varies with the size of the source values.
- The normalized version of covariance is the correlation coefficient, in which covariance is normalized between -1 and 1. For more information, see *CORREL Function*.

**Relevant terms:**

| Term | Description |
|------|-------------|
| Population | Population statistical functions are computed from all possible values. See *https://en.wikipedia.org/wiki/Statistical_population*. |
| Sample | Sample-based statistical functions are computed from a subset or sample of all values. See *https://en.wikipedia.org/wiki/Sampling_(statistics)*.<br><br>These function names include SAMP in their name.<br><br>**NOTE:** Statistical sampling has no relationship to the samples taken within the product. When statistical functions are computed during job execution, they are applied across the entire dataset. Sample method calculations are computed at that time. |

- This function is calculated across a sample of all values.
- For more information on a population version of this function, see *COVAR Function*.

**Wrangle vs. SQL:** This function is part of Wrangle , a proprietary data transformation language. Wrangle is not SQL. For more information, see *Wrangle Language*.

## Basic Usage

```
covarsamp(squareFootage,purchasePrice)
```

**Output:** Returns the covariance using the sample method between the values in the `squareFootage` column and the `purchasePrice` column.

## Syntax and Arguments

```
covarsamp(function_col_ref1,<span>function_col_ref2</span>) [group:group_col_ref] [limit:
limit_count]
```

| Argument | Required? | Data Type | Description |
|----------|-----------|-----------|-------------|
| function_col_ref1 | Y | string | Name of column that is the first input to the function |
| function_col_ref2 | Y | string | Name of column that is the second input to the function |

For more information on the `group` and `limit` parameters, see *Pivot Transform*.

For more information on syntax standards, see *Language Documentation Syntax Notes*.

**function_col_ref1, function_col_ref2**

Name of the column the values of which you want to calculate the covariance. Column must contain Integer or Decimal values.

- Literal values are not supported as inputs.
- Multiple columns and wildcards are not supported.

**Usage Notes:**

| Required? | Data Type | Example Value |
|---|---|---|
| Yes | String (column reference) | `myInputs` |

## Examples

> **Tip:** For additional examples, see *Common Tasks*.

This example illustrates statistical functions that can be applied across two columns of values.

**Functions**:

| Item | Description |
|---|---|
| CORREL Function | Computes the correlation coefficient between two columns. Source values can be of Integer or Decimal type. |
| COVAR Function | Computes the covariance between two columns using the population method. Source values can be of Integer or Decimal type. |
| COVARSAMP Function | Computes the covariance between two columns using the sample method. Source values can be of Integer or Decimal type. |
| ROUND Function | Rounds input value to the nearest integer. Input can be an Integer, a Decimal, a column reference, or an expression. Optional second argument can be used to specify the number of digits to which to round. |

**Source:**

The following table contains height in inches and weight in pounds for a set of students.

| Student | heightIn | weightLbs |
|---|---|---|
| 1 | 70 | 134 |
| 2 | 67 | 135 |
| 3 | 67 | 147 |
| 4 | 67 | 160 |
| 5 | 72 | 136 |
| 6 | 73 | 146 |
| 7 | 71 | 135 |
| 8 | 63 | 145 |
| 9 | 67 | 138 |

| | | |
|---|---|---|
| 10 | 66 | 138 |
| 11 | 71 | 161 |
| 12 | 70 | 131 |
| 13 | 74 | 131 |
| 14 | 67 | 157 |
| 15 | 73 | 161 |
| 16 | 70 | 133 |
| 17 | 63 | 132 |
| 18 | 64 | 153 |
| 19 | 64 | 156 |
| 20 | 72 | 154 |

### Transformation:

You can use the following transformations to calculate the correlation co-efficient, the covariance, and the sampling method covariance between the two data columns:

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | round(correl(heightIn, weightLbs), 3) |
| Parameter: New column name | 'corrHeightAndWeight' |

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | round(covar(heightIn, weightLbs), 3) |
| Parameter: New column name | 'covarHeightAndWeight' |

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | round(covarsamp(heightIn, weightLbs), 3) |
| Parameter: New column name | 'covarHeightAndWeight-Sample' |

### Results:

| Student | heightIn | weightLbs | covarHeightAndWeight-Sample | covarHeightAndWeight | corrHeightAndWeight |
|---|---|---|---|---|---|
| 1 | 70 | 134 | -2.876 | -2.732 | -0.074 |
| 2 | 67 | 135 | -2.876 | -2.732 | -0.074 |
| 3 | 67 | 147 | -2.876 | -2.732 | -0.074 |

| 4 | 67 | 160 | -2.876 | -2.732 | -0.074 |
|----|----|-----|--------|--------|--------|
| 5 | 72 | 136 | -2.876 | -2.732 | -0.074 |
| 6 | 73 | 146 | -2.876 | -2.732 | -0.074 |
| 7 | 71 | 135 | -2.876 | -2.732 | -0.074 |
| 8 | 63 | 145 | -2.876 | -2.732 | -0.074 |
| 9 | 67 | 138 | -2.876 | -2.732 | -0.074 |
| 10 | 66 | 138 | -2.876 | -2.732 | -0.074 |
| 11 | 71 | 161 | -2.876 | -2.732 | -0.074 |
| 12 | 70 | 131 | -2.876 | -2.732 | -0.074 |
| 13 | 74 | 131 | -2.876 | -2.732 | -0.074 |
| 14 | 67 | 157 | -2.876 | -2.732 | -0.074 |
| 15 | 73 | 161 | -2.876 | -2.732 | -0.074 |
| 16 | 70 | 133 | -2.876 | -2.732 | -0.074 |
| 17 | 63 | 132 | -2.876 | -2.732 | -0.074 |
| 18 | 64 | 153 | -2.876 | -2.732 | -0.074 |
| 19 | 64 | 156 | -2.876 | -2.732 | -0.074 |
| 20 | 72 | 154 | -2.876 | -2.732 | -0.074 |

# KTHLARGEST Function

Extracts the ranked value from the values in a column, where `k=1` returns the maximum value. The value for `k` must be between 1 and 1000, inclusive. Inputs can be Integer, Decimal, or Datetime.

For purposes of this calculation, two instances of the same value are treated as separate values. So, if your dataset contains three rows with column values `10` , `9` , and `9` , then `KTHLARGEST` returns `9` for `k=2` and `k=3` .

When used in a `pivot` transform, the function is computed for each instance of the value specified in the `group` parameter. See *Pivot Transform*.

Input column can be of Integer, Decimal, or Datetime type. Other values column are ignored. If a row contains a missing or null value, it is not factored into the calculation.

**Wrangle vs. SQL:** This function is part of Wrangle , a proprietary data transformation language. Wrangle is not SQL. For more information, see *Wrangle Language*.

## Basic Usage

```
kthlargest(myRating, 2)
```

**Output:** Returns the second highest value from the `myRating` column.

## Syntax and Arguments

```
kthlargest(function_col_ref, k_integer) [ group:group_col_ref] [limit:limit_count]
```

| Argument | Required? | Data Type | Description |
|---|---|---|---|
| function_col_ref | Y | string | Name of column to which to apply the function |
| k_integer | Y | integer (positive) | The ranking of the value to extract from the source column |

For more information on the `group` and `limit` parameters, see *Pivot Transform*.

For more information on syntax standards, see *Language Documentation Syntax Notes*.

**function_col_ref**

Name of the column the values of which you want to calculate the mean. Inputs must be Integer, Decimal, or Datetime values.

> **NOTE:** If the input is in Datetime type, the output is in unixtime format. You can wrap these outputs in the DATEFORMAT function to generate the results in the appropriate Datetime format. See *DATEFORMAT Function*.

- Literal values are not supported as inputs.
- Multiple columns and wildcards are not supported.

**Usage Notes:**

| Required? | Data Type | Example Value |
|---|---|---|
| Yes | String (column reference) | `myValues` |

## k_integer

Integer representing the ranking of the value to extract from the source column.

> **NOTE:** The value for `k` must be an integer between 1 and 1,000 inclusive.

- `k=1` represents the maximum value in the column.
- If k is greater than or equal to the number of values in the column, the minimum value is returned.
- Missing and null values are not factored into the ranking of `k`.

**Usage Notes:**

| Required? | Data Type | Example Value |
|---|---|---|
| Yes | Integer (positive) | 4 |

# Examples

> **Tip:** For additional examples, see *Common Tasks*.

This example explores how you can use aggregation functions to calculate rank of values in a column.

**Functions**:

| Item | Description |
|---|---|
| KTHLARGEST Function | Extracts the ranked value from the values in a column, where `k=1` returns the maximum value. The value for `k` must be between 1 and 1000, inclusive. Inputs can be Integer, Decimal, or Datetime. |
| KTHLARGESTUNIQUE Function | Extracts the ranked unique value from the values in a column, where `k=1` returns the maximum value. The value for `k` must be between 1 and 1000, inclusive. Inputs can be Integer, Decimal, or Datetime. |

**Source:**

You have a set of student test scores:

| Student | Score |
|---|---|
| Anna | 84 |
| Ben | 71 |
| Caleb | 76 |
| Danielle | 87 |
| Evan | 85 |
| Faith | 92 |

| | |
|---|---|
| Gabe | 87 |
| Hannah | 99 |
| Ian | 73 |
| Jane | 68 |

**Transformation:**

You can use the following transformations to extract the 1st through 4th-ranked scores on the test:

| **Transformation Name** | New formula |
|---|---|
| **Parameter: Formula type** | Single row formula |
| **Parameter: Formula** | KTHLARGEST(Score, 1) |
| **Parameter: New column name** | '1st' |

| **Transformation Name** | New formula |
|---|---|
| **Parameter: Formula type** | Single row formula |
| **Parameter: Formula** | KTHLARGEST(Score, 2) |
| **Parameter: New column name** | '2nd' |

| **Transformation Name** | New formula |
|---|---|
| **Parameter: Formula type** | Single row formula |
| **Parameter: Formula** | KTHLARGEST(Score, 3) |
| **Parameter: New column name** | '3rd' |

| **Transformation Name** | New formula |
|---|---|
| **Parameter: Formula type** | Single row formula |
| **Parameter: Formula** | KTHLARGEST(Score, 4) |
| **Parameter: New column name** | '4th' |

| **Transformation Name** | New formula |
|---|---|
| **Parameter: Formula type** | Single row formula |
| **Parameter: Formula** | KTHLARGESTUNIQUE(Score, 3) |
| **Parameter: New column name** | '3rdUnique' |

| **Transformation Name** | New formula |
|---|---|
| **Parameter: Formula type** | Single row formula |

| Parameter: Formula | KTHLARGESTUNIQUE(Score, 4) |
|---|---|
| Parameter: New column name | '4thUnique' |

**Results:**

When you reorganize the columns, the dataset might look like the following:

| Student | Score | 1st | 2nd | 3rd | 4th | 3rdUnique | 4thUnique |
|---|---|---|---|---|---|---|---|
| Anna | 84 | 99 | 92 | 87 | 87 | 87 | 85 |
| Ben | 71 | 99 | 92 | 87 | 87 | 87 | 85 |
| Caleb | 76 | 99 | 92 | 87 | 87 | 87 | 85 |
| Danielle | 87 | 99 | 92 | 87 | 87 | 87 | 85 |
| Evan | 85 | 99 | 92 | 87 | 87 | 87 | 85 |
| Faith | 92 | 99 | 92 | 87 | 87 | 87 | 85 |
| Gabe | 87 | 99 | 92 | 87 | 87 | 87 | 85 |
| Hannah | 99 | 99 | 92 | 87 | 87 | 87 | 85 |
| Ian | 73 | 99 | 92 | 87 | 87 | 87 | 85 |
| Jane | 68 | 99 | 92 | 87 | 87 | 87 | 85 |

Notes:

- The value `87` is both the third and fourth scores.
    - For the `KTHLARGEST` function, it is the output for the third and fourth ranking.
    - For the `KTHLARGESTUNIQUE` function, it is the output for the third ranking only.

# KTHLARGESTIF Function

**Contents:**

Extracts the ranked value from the values in a column, where `k=1` returns the maximum value, when a specified condition is met. The value for `k` must be between 1 and 1000, inclusive. Inputs can be Integer, Decimal, or Datetime.

`KTHLARGESTIF` calculations are filtered by a conditional applied to the group.

For purposes of this calculation, two instances of the same value are treated as separate values. So, if your dataset contains three rows with column values `10`, `9`, and `9`, then `KTHLARGEST` returns `9` for `k=2` and `k=3`.

Input column can be of Integer, Decimal, or Datetime type. Other values column are ignored. If a row contains a missing or null value, it is not factored into the calculation.

> **NOTE:** When added to a transformation, this function is applied to the current sample. If you change your sample or run the job, the computed values for this function are updated. Transformations that change the number of rows in subsequent recipe steps do not affect the values computed for this step.

To perform a simple kth largest calculation without conditionals, use the `KTHLARGEST` function. See *KTHLARGEST Function*.

**Wrangle vs. SQL:** This function is part of Wrangle , a proprietary data transformation language. Wrangle is not SQL. For more information, see *Wrangle Language*.

## Basic Usage

```
kthlargestif(POS_Sales, 1, DayOfWeek == &apos;Saturday&apos;)
```

**Output:** Returns the top value (rank=1) from the `POS_Sales` column when the `DayOfWeek` value is `Saturday`.

## Syntax and Arguments

```
kthlargestif(col_ref, limit, test_expression) [group:group_col_ref] [limit:limit_count]
```

| Argument | Required? | Data Type | Description |
|---|---|---|---|
| col_ref | Y | string | Reference to the column you wish to evaluate. |
| k_integer | Y | integer | The ranking of the value to extract from the source column |
| test_expression | Y | string | Expression that is evaluated. Must resolve to `true` or `false` |

For more information on syntax standards, see *Language Documentation Syntax Notes*.

For more information on the `group` and `limit` parameter, see *Pivot Transform*.

### col_ref

Name of the column whose values you wish to use in the calculation. Inputs must be Integer, Decimal, or Datetime values.

> **NOTE:** If the input is in Datetime type, the output is in unixtime format. You can wrap these outputs in the DATEFORMAT function to output the results in the appropriate Datetime format. See *DATEFORMAT Function*.

**Usage Notes:**

| Required? | Data Type | Example Value |
|---|---|---|
| Yes | String that corresponds to the name of the column | `myValues` |

### k_integer

Integer representing the ranking of the value to extract from the source column.

> **NOTE:** The value for `k` must be an integer between 1 and 1,000 inclusive.

- `k=1` represents the maximum value in the column.
- If k is greater than or equal to the number of values in the column, the minimum value is returned.
- Missing and null values are not factored into the ranking of `k`.

### test_expression

This parameter contains the expression to evaluate. This expression must resolve to a Boolean (`true` or `false`) value.

**Usage Notes:**

| Required? | Data Type | Example Value |
|---|---|---|
| Yes | String expression that evaluates to `true` or `false` | `(LastName == 'Mouse' && FirstName == 'Mickey')` |

## Examples

> **Tip:** For additional examples, see *Common Tasks*.

**Example - Second-most measurements for a specific city**

This example illustrates how to use the conditional ranking functions.

**Functions:**

| Item | Description |
|------|-------------|
| KTHLARGESTIF Function | Extracts the ranked value from the values in a column, where $k=1$ returns the maximum value, when a specified condition is met. The value for $k$ must be between 1 and 1000, inclusive. Inputs can be Integer, Decimal, or Datetime. |
| KTHLARGESTUNIQUEIF Function | Extracts the ranked unique value from the values in a column, where $k=1$ returns the maximum value, when a specified condition is met. The value for $k$ must be between 1 and 1000, inclusive. Inputs can be Integer, Decimal, or Datetime. |

**Source:**

Here is some example weather data:

| date | city | rain_cm | temp_C | wind_mph |
|------|------|---------|--------|----------|
| 1/23/17 | Valleyville | 0.00 | 12.8 | 8.8 |
| 1/23/17 | Center Town | 0.31 | 9.4 | 5.3 |
| 1/23/17 | Magic Mountain | 0.00 | 0.0 | 7.3 |
| 1/24/17 | Valleyville | 0.25 | 17.2 | 3.3 |
| 1/24/17 | Center Town | 0.54 | 1.1 | 7.6 |
| 1/24/17 | Magic Mountain | 0.32 | 5.0 | 8.8 |
| 1/25/17 | Valleyville | 0.02 | 3.3 | 6.8 |
| 1/25/17 | Center Town | 0.83 | 3.3 | 5.1 |
| 1/25/17 | Magic Mountain | 0.59 | -1.7 | 6.4 |
| 1/26/17 | Valleyville | 1.08 | 15.0 | 4.2 |
| 1/26/17 | Center Town | 0.96 | 6.1 | 7.6 |
| 1/26/17 | Magic Mountain | 0.77 | -3.9 | 3.0 |
| 1/27/17 | Valleyville | 1.00 | 7.2 | 2.8 |
| 1/27/17 | Center Town | 1.32 | 20.0 | 0.2 |
| 1/27/17 | Magic Mountain | 0.77 | 5.6 | 5.2 |
| 1/28/17 | Valleyville | 0.12 | -6.1 | 5.1 |
| 1/28/17 | Center Town | 0.14 | 5.0 | 4.9 |
| 1/28/17 | Magic Mountain | 1.50 | 1.1 | 0.4 |
| 1/29/17 | Valleyville | 0.36 | 13.3 | 7.3 |
| 1/29/17 | Center Town | 0.75 | 6.1 | 9.0 |
| 1/29/17 | Magic Mountain | 0.60 | 3.3 | 6.0 |

**Transformation:**

In this case, you want to find out the second-most measures for rain, temperature, and wind in Center Town for the week.

| Transformation Name | Pivot columns |
|---------------------|---------------|
| **Parameter: Values** | `KTHLARGESTIF(rain_cm,2,city == 'Center Town')` |
| **Parameter: Max number of columns to create** | `1` |

You can see in the preview that the value is `1.32`. Before adding it to your recipe, you change the step to the following:

| Transformation Name | `Pivot columns` |
|---|---|
| **Parameter: Values** | `KTHLARGESTIF(temp_C,2,city == 'Center Town')` |
| **Parameter: Max number of columns to create** | `1` |

The value is `20`.

For wind, you modify it to be the following, capturing the third-ranked value:

| Transformation Name | `Pivot columns` |
|---|---|
| **Parameter: Values** | `KTHLARGESTIF(wind_mph,3,city == 'Center Town')` |
| **Parameter: Max number of columns to create** | `1` |

In the results, you notice that there are two values for `8.8`. So you change the function to use the `KTHLARGESTUNIQUEIF` function instead:

| Transformation Name | `Pivot columns` |
|---|---|
| **Parameter: Values** | `KTHLARGESTUNIQUEIF(wind_mph,3,city == 'Center Town')` |
| **Parameter: Max number of columns to create** | `1` |

The result value is `7.6`. Note that this value appears twice, so if you change the rank parameter in the above transformation to `4`, the results would return a different unique ranked value (`7.3`).

**Results:**

You can choose to add any of these steps to generate an aggregated result. As an alternative, you can use a `derive` transform to insert these calculated results into new columns.

# KTHLARGESTUNIQUE Function

Extracts the ranked unique value from the values in a column, where `k=1` returns the maximum value. The value for `k` must be between 1 and 1000, inclusive. Inputs can be Integer, Decimal, or Datetime.

For purposes of this calculation, two instances of the same value are treated as the same value of `k`. So, if your dataset contains four rows with column values `10` , `9` , `9` , and `8`, then `KTHLARGEST` returns `9` for `k=2` and `8` for `k=3` .

- For a non-unique version of this function, see *KTHLARGEST Function*.

When used in a `pivot` transform, the function is computed for each instance of the value specified in the `group` parameter. See *Pivot Transform*.

Input column can be of Integer, Decimal, or Datetime type. Other values column are ignored. If a row contains a missing or null value, it is not factored into the calculation.

**Wrangle vs. SQL:** This function is part of Wrangle , a proprietary data transformation language. Wrangle is not SQL. For more information, see *Wrangle Language*.

## Basic Usage

```
kthlargestunique(myRating, 3)
```

**Output:** Returns the third highest unique value from the `myRating` column.

## Syntax and Arguments

```
kthlargestunique(function_col_ref, k_integer) [ group:group_col_ref] [limit:limit_count]
```

| Argument | Required? | Data Type | Description |
|---|---|---|---|
| function_col_ref | Y | string | Name of column to which to apply the function |
| k_integer | Y | integer (positive) | The ranking of the unique value to extract from the source column |

For more information on the `group` and `limit` parameters, see *Pivot Transform*.

For more information on syntax standards, see *Language Documentation Syntax Notes*.

**function_col_ref**

Name of the column the values of which you want to calculate the mean. Inputs must be Integer, Decimal, or Datetime values.

> **NOTE:** If the input is in Datetime type, the output is in unixtime format. You can wrap these outputs in the DATEFORMAT function to output the results in the appropriate Datetime format. See *DATEFORMAT Function*.

- Literal values are not supported as inputs.
- Multiple columns and wildcards are not supported.

**Usage Notes:**

| Required? | Data Type | Example Value |
|---|---|---|
| Yes | String (column reference) | `myValues` |

### k_integer

Integer representing the ranking of the unique value to extract from the source column. Duplicate values are treated as a single value for purposes of this function's calculation.

> **NOTE:** The value for $k$ must be an integer between 1 and 1,000 inclusive.

- `k=1` represents the maximum value in the column.
- If k is greater than or equal to the number of values in the column, the minimum value is returned.
- Missing and null values are not factored into the ranking of $k$.

**Usage Notes:**

| Required? | Data Type | Example Value |
|---|---|---|
| Yes | Integer (positive) | 4 |

## Examples

> **Tip:** For additional examples, see *Common Tasks*.

This example explores how you can use aggregation functions to calculate rank of values in a column.

**Functions**:

| Item | Description |
|---|---|
| KTHLARGEST Function | Extracts the ranked value from the values in a column, where $k=1$ returns the maximum value. The value for $k$ must be between 1 and 1000, inclusive. Inputs can be Integer, Decimal, or Datetime. |
| KTHLARGESTUNIQUE Function | Extracts the ranked unique value from the values in a column, where $k=1$ returns the maximum value. The value for $k$ must be between 1 and 1000, inclusive. Inputs can be Integer, Decimal, or Datetime. |

**Source:**

You have a set of student test scores:

| Student | Score |
|---|---|
| Anna | 84 |
| Ben | 71 |
| Caleb | 76 |
| Danielle | 87 |

| | |
|---|---|
| Evan | 85 |
| Faith | 92 |
| Gabe | 87 |
| Hannah | 99 |
| Ian | 73 |
| Jane | 68 |

**Transformation:**

You can use the following transformations to extract the 1st through 4th-ranked scores on the test:

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | KTHLARGEST(Score, 1) |
| Parameter: New column name | '1st' |

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | KTHLARGEST(Score, 2) |
| Parameter: New column name | '2nd' |

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | KTHLARGEST(Score, 3) |
| Parameter: New column name | '3rd' |

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | KTHLARGEST(Score, 4) |
| Parameter: New column name | '4th' |

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | KTHLARGESTUNIQUE(Score, 3) |
| Parameter: New column name | '3rdUnique' |

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | KTHLARGESTUNIQUE(Score, 4) |
| Parameter: New column name | '4thUnique' |

**Results:**

When you reorganize the columns, the dataset might look like the following:

| Student | Score | 1st | 2nd | 3rd | 4th | 3rdUnique | 4thUnique |
|---|---|---|---|---|---|---|---|
| Anna | 84 | 99 | 92 | 87 | 87 | 87 | 85 |
| Ben | 71 | 99 | 92 | 87 | 87 | 87 | 85 |
| Caleb | 76 | 99 | 92 | 87 | 87 | 87 | 85 |
| Danielle | 87 | 99 | 92 | 87 | 87 | 87 | 85 |
| Evan | 85 | 99 | 92 | 87 | 87 | 87 | 85 |
| Faith | 92 | 99 | 92 | 87 | 87 | 87 | 85 |
| Gabe | 87 | 99 | 92 | 87 | 87 | 87 | 85 |
| Hannah | 99 | 99 | 92 | 87 | 87 | 87 | 85 |
| Ian | 73 | 99 | 92 | 87 | 87 | 87 | 85 |
| Jane | 68 | 99 | 92 | 87 | 87 | 87 | 85 |

Notes:

- The value 87 is both the third and fourth scores.
  - For the KTHLARGEST function, it is the output for the third and fourth ranking.
  - For the KTHLARGESTUNIQUE function, it is the output for the third ranking only.

# KTHLARGESTUNIQUEIF Function

**Contents:**

Extracts the ranked unique value from the values in a column, where `k=1` returns the maximum value, when a specified condition is met. The value for `k` must be between 1 and 1000, inclusive. Inputs can be Integer, Decimal, or Datetime.

`KTHLARGESTUNIQUEIF` calculations are filtered by a conditional applied to the group.

For purposes of this calculation, two instances of the same value are treated as the same value of `k`. So, if your dataset contains four rows with column values `10` , `9` , `9` , and `8`, the the function returns `9` for `k=2` and `8` for `k=3`.

Input column can be of Integer, Decimal or Datetime type. Other values column are ignored. If a row contains a missing or null value, it is not factored into the calculation.

> **NOTE:** When added to a transformation, this function is applied to the current sample. If you change your sample or run the job, the computed values for this function are updated. Transformations that change the number of rows in subsequent recipe steps do not affect the values computed for this step.

To perform a simple kth largest unique calculation without conditionals, use the `KTHLARGESTUNIQUE` function. See *KTHLARGESTUNIQUE Function*.

**Wrangle vs. SQL:** This function is part of Wrangle , a proprietary data transformation language. Wrangle is not SQL. For more information, see *Wrangle Language*.

## Basic Usage

```
kthlargestuniqueif(POS_Sales, 2, DayOfWeek == &apos;Saturday&apos;)
```

**Output:** Returns the secondmost value (rank=2) from the `POS_Sales` column when the `DayOfWeek` value is `Saturday`.

## Syntax and Arguments

```
kthlargestuniqueif(col_ref, limit, test_expression) [group:group_col_ref] [limit:
limit_count]
```

| Argument | Required? | Data Type | Description |
|----------|-----------|-----------|-------------|
| col_ref | Y | string | Reference to the column you wish to evaluate. |
| k_integer | Y | integer | The ranking of the value to extract from the source column |

| test_expression | Y | string | Expression that is evaluated. Must resolve to `true` or `false` |
|---|---|---|---|

For more information on syntax standards, see *Language Documentation Syntax Notes*.

For more information on the `group` and `limit` parameter, see *Pivot Transform*.

### col_ref

Name of the column whose values you wish to use in the calculation. Inputs must be Integer, Decimal, or Datetime values.

> **NOTE:** If the input is in Datetime type, the output is in unixtime format. You can wrap these outputs in the DATEFORMAT function to output the results in the appropriate Datetime format. See *DATEFORMAT Function*.

**Usage Notes:**

| Required? | Data Type | Example Value |
|---|---|---|
| Yes | String that corresponds to the name of the column | `myValues` |

### k_integer

Integer representing the unique ranking of the value to extract from the source column.

> **NOTE:** The value for `k` must be an integer between 1 and 1,000 inclusive.

- `k=1` represents the maximum value in the column.
- If k is greater than or equal to the number of values in the column, the minimum value is returned.
- Missing and null values are not factored into the ranking of `k`.

### test_expression

This parameter contains the expression to evaluate. This expression must resolve to a Boolean (`true` or `false`) value.

**Usage Notes:**

| Required? | Data Type | Example Value |
|---|---|---|
| Yes | String expression that evaluates to `true` or `false` | `(LastName == 'Mouse' && FirstName == 'Mickey')` |

## Examples

> **Tip:** For additional examples, see *Common Tasks*.

**Example - Second-most measurements for a specific city**

This example illustrates how to use the conditional ranking functions.

**Functions:**

| Item | Description |
|---|---|
| KTHLARGESTIF Function | Extracts the ranked value from the values in a column, where $k=1$ returns the maximum value, when a specified condition is met. The value for $k$ must be between 1 and 1000, inclusive. Inputs can be Integer, Decimal, or Datetime. |
| KTHLARGESTUNIQUEIF Function | Extracts the ranked unique value from the values in a column, where $k=1$ returns the maximum value, when a specified condition is met. The value for $k$ must be between 1 and 1000, inclusive. Inputs can be Integer, Decimal, or Datetime. |

**Source:**

Here is some example weather data:

| date | city | rain_cm | temp_C | wind_mph |
|---|---|---|---|---|
| 1/23/17 | Valleyville | 0.00 | 12.8 | 8.8 |
| 1/23/17 | Center Town | 0.31 | 9.4 | 5.3 |
| 1/23/17 | Magic Mountain | 0.00 | 0.0 | 7.3 |
| 1/24/17 | Valleyville | 0.25 | 17.2 | 3.3 |
| 1/24/17 | Center Town | 0.54 | 1.1 | 7.6 |
| 1/24/17 | Magic Mountain | 0.32 | 5.0 | 8.8 |
| 1/25/17 | Valleyville | 0.02 | 3.3 | 6.8 |
| 1/25/17 | Center Town | 0.83 | 3.3 | 5.1 |
| 1/25/17 | Magic Mountain | 0.59 | -1.7 | 6.4 |
| 1/26/17 | Valleyville | 1.08 | 15.0 | 4.2 |
| 1/26/17 | Center Town | 0.96 | 6.1 | 7.6 |
| 1/26/17 | Magic Mountain | 0.77 | -3.9 | 3.0 |
| 1/27/17 | Valleyville | 1.00 | 7.2 | 2.8 |
| 1/27/17 | Center Town | 1.32 | 20.0 | 0.2 |
| 1/27/17 | Magic Mountain | 0.77 | 5.6 | 5.2 |
| 1/28/17 | Valleyville | 0.12 | -6.1 | 5.1 |
| 1/28/17 | Center Town | 0.14 | 5.0 | 4.9 |
| 1/28/17 | Magic Mountain | 1.50 | 1.1 | 0.4 |
| 1/29/17 | Valleyville | 0.36 | 13.3 | 7.3 |
| 1/29/17 | Center Town | 0.75 | 6.1 | 9.0 |
| 1/29/17 | Magic Mountain | 0.60 | 3.3 | 6.0 |

**Transformation:**

In this case, you want to find out the second-most measures for rain, temperature, and wind in Center Town for the week.

| Transformation Name | Pivot columns |
|---|---|
| Parameter: Values | KTHLARGESTIF(rain_cm,2,city == 'Center Town') |

| Parameter: Max number of columns to create | 1 |
|---|---|

You can see in the preview that the value is `1.32`. Before adding it to your recipe, you change the step to the following:

| Transformation Name | `Pivot columns` |
|---|---|
| Parameter: Values | `KTHLARGESTIF(temp_C,2,city == 'Center Town')` |
| Parameter: Max number of columns to create | 1 |

The value is `20`.

For wind, you modify it to be the following, capturing the third-ranked value:

| Transformation Name | `Pivot columns` |
|---|---|
| Parameter: Values | `KTHLARGESTIF(wind_mph,3,city == 'Center Town')` |
| Parameter: Max number of columns to create | 1 |

In the results, you notice that there are two values for `8.8`. So you change the function to use the `KTHLARGESTUNIQUEIF` function instead:

| Transformation Name | `Pivot columns` |
|---|---|
| Parameter: Values | `KTHLARGESTUNIQUEIF(wind_mph,3,city == 'Center Town')` |
| Parameter: Max number of columns to create | 1 |

The result value is `7.6`. Note that this value appears twice, so if you change the rank parameter in the above transformation to `4`, the results would return a different unique ranked value (`7.3`).

**Results:**

You can choose to add any of these steps to generate an aggregated result. As an alternative, you can use a `derive` transform to insert these calculated results into new columns.

# LIST Function

**Contents:**

---

Extracts the set of values from a column into an array stored in a new column. This function is typically part of an aggregation.

> **Tip:** To generate unique values for the list, apply the ARRAYUNIQUE function in the next step after this one. See *ARRAYUNIQUE Function*.

Input column can be of any type.

- By default, the list is limited to `1000` values. To change the maximum number of values, specify a value for the `limit` parameter.
- This function is intended to be used as part of an aggregation to return the distinct set of values by group. See *Pivot Transform*.

For a version of this function computed over a rolling window of rows, see *ROLLINGLIST Function*.

**Wrangle vs. SQL:** This function is part of Wrangle , a proprietary data transformation language. Wrangle is not SQL. For more information, see *Wrangle Language*.

## Basic Usage

```
<span>list(Name, 500)</span>
```

**Output:** Returns an array of all values (up to a count of `500`) from the `Name` column for each `Month` value.

## Syntax and Arguments

```
list(function_col_ref, [limit_int]) [ group:group_col_ref] [limit:limit_count]
```

| Argument | Required? | Data Type | Description |
|---|---|---|---|
| function_col_ref | Y | string | Name of column to which to apply the function |
| limit_int | N | integer (positive) | Maximum number of values to extract into the list array. From `1` to `1000`. |

For more information on the `group` and `limit` parameters, see *Pivot Transform*.

For more information on syntax standards, see *Language Documentation Syntax Notes*.

**function_col_ref**

Name of the column from which to extract the list of values based on the grouping.

- Literal values are not supported as inputs.
- Multiple columns and wildcards are not supported.

**Usage Notes:**

| Required? | Data Type | Example Value |
|---|---|---|
| Yes | String (column reference) | `myValues` |

**limit_int**

Non-negative integer that defines the maximum number of values to extract into the list array.

> **NOTE:** If specified, this value must between 1 and 1000, inclusive.

> **NOTE:** Do not use the limiting argument in a `LIST` function call on a flat aggregate, in which all values in a column have been inserted into a single cell. In this case, you might be able to use the limit argument if you also specify a `group` parameter. Misuse of the `LIST` function can cause the application to crash.

**Usage Notes:**

| Required? | Data Type | Example Value |
|---|---|---|
| No | Integer | `50` |

## Examples

> **Tip:** For additional examples, see *Common Tasks*.

**Example - Colors sold this month**

This example demonstrates you to extract values from one column of an array into a new column.

**Functions**:

| Item | Description |
|---|---|
| LIST Function | Extracts the set of values from a column into an array stored in a new column. This function is typically part of an aggregation. |
| UNIQUE Function | Extracts the set of unique values from a column into an array stored in a new column. This function is typically part of an aggregation. |
| DATEFORMAT Function | Formats a specified Datetime set of values according to the specified date format. Source values can be a reference to a column containing Datetime values. |

You have the following set of orders for two months, and you are interested in identifying the set of colors that have been sold for each product for each month and the total quantity of product sold for each month.

**Source:**

| OrderId | Date | Item | Qty | Color |
|---------|---------|-------|-----|--------|
| 1001 | 1/15/15 | Pants | 1 | red |
| 1002 | 1/15/15 | Shirt | 2 | green |
| 1003 | 1/15/15 | Hat | 3 | blue |
| 1004 | 1/16/15 | Shirt | 4 | yellow |
| 1005 | 1/16/15 | Hat | 5 | red |
| 1006 | 1/20/15 | Pants | 6 | green |
| 1007 | 1/15/15 | Hat | 7 | blue |
| 1008 | 4/15/15 | Shirt | 8 | yellow |
| 1009 | 4/15/15 | Shoes | 9 | brown |
| 1010 | 4/16/15 | Pants | 1 | red |
| 1011 | 4/16/15 | Hat | 2 | green |
| 1012 | 4/16/15 | Shirt | 3 | blue |
| 1013 | 4/20/15 | Shoes | 4 | black |
| 1014 | 4/20/15 | Hat | 5 | blue |
| 1015 | 4/20/15 | Pants | 6 | black |

**Transformation:**

To track by month, you need a column containing the month value extracted from the date:

| Transformation Name | `Edit column with formula` |
|---------------------|----------------------------|
| **Parameter: Columns** | `Date` |
| **Parameter: Formula** | `DATEFORMAT(Date, 'MMM yyyy')` |

You can use the following transform to check the list of unique values among the colors:

| Transformation Name | `Pivot columns` |
|---------------------|-----------------|
| **Parameter: Row labels** | `Date` |
| **Parameter: Values** | `unique(Color, 1000)` |
| **Parameter: Max number of columns to create** | `10` |

| Date | unique_Color |
|----------|--------------|
| Jan 2015 | ["green","blue","red","yellow"] |
| Apr 2015 | ["brown","blue","red","yellow","black","green"] |

Delete the above transform.

You can aggregate the data in your dataset, grouped by the reformatted `Date` values, and apply the `LIST` function to the `Color` column. In the same aggregation, you can include a summation function for the `Qty` column:

| Transformation Name | Pivot columns |
|---|---|
| Parameter: Row labels | Date |
| Parameter: Values | list(Color, 1000),sum(Qty) |
| Parameter: Max number of columns to create | 10 |

**Results:**

| Date | list_Color | sum_Qty |
|---|---|---|
| Jan 2015 | ["green","blue","blue","red","green","red","yellow"] | 28 |
| Apr 2015 | ["brown","blue","red","yellow","black","blue","black","green"] | 38 |

# LISTIF Function

**Contents:**

Returns list of all values in a column for rows that match a specified condition.

> **NOTE:** When added to a transformation, this function is applied to the current sample. If you change your sample or run the job, the computed values for this function are updated. Transformations that change the number of rows in subsequent recipe steps do not affect the values computed for this step.

To perform a simple extraction of values without conditionals, use the LIST function. See *LIST Function*.

**Wrangle vs. SQL:** This function is part of Wrangle , a proprietary data transformation language. Wrangle is not SQL. For more information, see *Wrangle Language*.

## Basic Usage

```
listif(hotChocolateLiters, 500, temperature &lt; 0)
```

**Output:** Returns the values from the `hotChocolateLiters` column when the `temperature` value is less than 0. Maximum number of values is 500.

## Syntax and Arguments

```
listif(col_ref, limit, test_expression) [group:group_col_ref] [limit:limit_count]
```

| Argument | Required? | Data Type | Description |
|---|---|---|---|
| col_ref | Y | string | Reference to the column you wish to evaluate. |
| limit_int | N | integer | Maximum number of values to extract into the list array. From $1$ to $1000$. |
| test_expression | Y | string | Expression that is evaluated. Must resolve to `true` or `false` |

For more information on syntax standards, see *Language Documentation Syntax Notes*.

For more information on the `group` and `limit` parameters, see *Pivot Transform*.

**col_ref**

Name of the column whose values you wish to use in the calculation. Column must be a numeric (Integer or Decimal) type.

**Usage Notes:**

| Required? | Data Type | Example Value |
|---|---|---|
| Yes | String that corresponds to the name of the column | `myValues` |

**limit_int**

Non-negative integer that defines the maximum number of values to extract into the list array.

> **NOTE:** If specified, this value must between 1 and 1000, inclusive.

> **NOTE:** Do not use the limiting argument in a `LISTIF` function call on a flat aggregate, in which all values in a column have been inserted into a single cell. In this case, you might be able to use the limit argument if you also specify a `group` parameter. Misuse of the `LISTIF` function can cause the application to crash.

**test_expression**

This parameter contains the expression to evaluate. This expression must resolve to a Boolean (`true` or `false`) value.

**Usage Notes:**

| Required? | Data Type | Example Value |
|---|---|---|
| Yes | String expression that evaluates to `true` or `false` | `(LastName == 'Mouse' && FirstName == 'Mickey')` |

## Examples

> **Tip:** For additional examples, see *Common Tasks*.

### Example - ANYIF and LISTIF Functions

This example illustrates you to identify and list all values within a group that meet a specified condition.

**Functions**:

| Item | Description |
|---|---|
| ANYIF Function | Selects a single non-null value from rows in each group that meet a specific condition. |
| LISTIF Function | Returns list of all values in a column for rows that match a specified condition. |
| WEEKDAY Function | Derives the numeric value for the day of the week (`1`, `2`, etc.). Input must be a reference to a column containing Datetime values. |

**Source:**

The following data identifies sales figures by salespeople for a week:

| EmployeeId | Date | Sales |
|---|---|---|
| S001 | 1/23/17 | 25 |
| S002 | 1/23/17 | 40 |
| S003 | 1/23/17 | 48 |
| S001 | 1/24/17 | 81 |
| S002 | 1/24/17 | 11 |
| S003 | 1/24/17 | 25 |
| S001 | 1/25/17 | 9 |
| S002 | 1/25/17 | 40 |
| S003 | 1/25/17 | |
| S001 | 1/26/17 | 77 |
| S002 | 1/26/17 | 83 |
| S003 | 1/26/17 | |
| S001 | 1/27/17 | 17 |
| S002 | 1/27/17 | 71 |
| S003 | 1/27/17 | 29 |
| S001 | 1/28/17 | |
| S002 | 1/28/17 | |
| S003 | 1/28/17 | 14 |
| S001 | 1/29/17 | 2 |
| S002 | 1/29/17 | 7 |
| S003 | 1/29/17 | 99 |

**Transformation:**

In this example, you are interested in the high performers. A good day in sales is one in which an individual sells more than 80 units. First, you want to identify the day of week:

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | WEEKDAY(Date) |
| Parameter: New column name | 'DayOfWeek' |

Values greater than 5 in `DayOfWeek` are weekend dates. You can use the following to identify if anyone reached this highwater marker during the workweek (non-weekend):

| Transformation Name | Pivot columns |
|---|---|
| Parameter: Rows labels | EmployeeId,Date |
| Parameter: Values | ANYIF(Sales, (Sales > 80 && DayOfWeek < 6)) |
| Parameter: Max number of columns to create | 1 |

Before adding the step to the recipe, you take note of the individuals who reached this mark in the `anyif_Sales` column for special recognition.

Now, you want to find out sales for individuals during the week. You can use the following to filter the data to show only for weekdays:

| Transformation Name | Pivot columns |
|---|---|
| Parameter: Rows labels | EmployeeId,Date |
| Parameter: Values | LISTIF(Sales, 1000, (DayOfWeek < 6)) |
| Parameter: Max number of columns to create | 1 |

To clean up, you might select and replace the following values in the listif_Sales column with empty strings:

```
["
"]
[]
```

**Results:**

| EmployeeId | Date | listif_Sales |
|---|---|---|
| S001 | 1/23/17 | 25 |
| S002 | 1/23/17 | 40 |
| S003 | 1/23/17 | 48 |
| S001 | 1/24/17 | 81 |
| S002 | 1/24/17 | 11 |
| S003 | 1/24/17 | 25 |
| S001 | 1/25/17 | 40 |
| S002 | 1/25/17 | |
| S003 | 1/25/17 | 66 |
| S001 | 1/26/17 | 77 |
| S002 | 1/26/17 | 83 |
| S003 | 1/26/17 | |
| S001 | 1/27/17 | 17 |
| S002 | 1/27/17 | 71 |
| S003 | 1/27/17 | 29 |
| S001 | 1/28/17 | |

| | | |
|---|---|---|
| S002 | 1/28/17 | |
| S003 | 1/28/17 | |
| S001 | 1/29/17 | |
| S002 | 1/29/17 | |
| S003 | 1/29/17 | |

# MAX Function

Computes the maximum value found in all row values in a column. Inputs can be Integer, Decimal, or Datetime.

- When used in a `pivot` transform, the function is computed for each instance of the value specified in the `group` parameter. See *Pivot Transform*.
- If a row contains a missing or null value, it is not factored into the calculation.
- If no numeric values are found in the source column, the function returns a null value.

For a version of this function computed over a rolling window of rows, see *ROLLINGMAX Function*.

Datetime inputs to this function return Unixtime values.

- These values can be wrapped in a DATEFORMAT function. See *DATEFORMAT Function*.
- For a date-native version of this function, see *MAXDATE Function*.

**Wrangle vs. SQL:** This function is part of Wrangle , a proprietary data transformation language. Wrangle is not SQL. For more information, see *Wrangle Language*.

## Basic Usage

```
max(myRating)
```

**Output:** Returns the maximum value of the `myRating` column.

## Syntax and Arguments

```
max(function_col_ref) [group:group_col_ref] [limit:limit_count]
```

| Argument | Required? | Data Type | Description |
|---|---|---|---|
| function_col_ref | Y | string | Name of column to which to apply the function |

For more information on the `group` and `limit` parameters, see *Pivot Transform*.

For more information on syntax standards, see *Language Documentation Syntax Notes*.

**function_col_ref**

Name of the column the values of which you want to calculate the maximum. Inputs must be Integer, Decimal, or Datetime values.

> **NOTE:** If the input is in Datetime type, the output is in unixtime format. You can wrap these outputs in the DATEFORMAT function to generate the results in the appropriate Datetime format. See *DATEFORMAT Function*.

- Literal values are not supported as inputs.
- Multiple columns and wildcards are not supported.

**Usage Notes:**

| Required? | Data Type | Example Value |
|---|---|---|
| Yes | String (column reference) | `myValues` |

## Examples

> **Tip:** For additional examples, see *Common Tasks*.

This example illustrates how you can apply statistical functions to your dataset. Calculations include average (mean), max, min, standard deviation, and variance.

**Functions:**

| Item | Description |
|---|---|
| AVERAGE Function | Computes the average (mean) from all row values in a column or group. Input column can be of Integer or Decimal. |
| MIN Function | Computes the minimum value found in all row values in a column. Input column can be of Integer, Decimal or Datetime. |
| MAX Function | Computes the maximum value found in all row values in a column. Inputs can be Integer, Decimal, or Datetime. |
| VAR Function | Computes the variance among all values in a column. Input column can be of Integer or Decimal. If no numeric values are detected in the input column, the function returns $0$. |
| STDEV Function | Computes the standard deviation across all column values of Integer or Decimal type. |
| NUMFORMAT Function | Formats a numeric set of values according to the specified number formatting. Source values can be a literal numeric value, a function returning a numeric value, or reference to a column containing an Integer or Decimal values. |
| MODE Function | Computes the mode (most frequent value) from all row values in a column, according to their grouping. Input column can be of Integer, Decimal, or Datetime type. |

**Source:**

Students took a test and recorded the following scores. You want to perform some statistical analysis on them:

| Student | Score |
|---|---|
| Anna | 84 |
| Ben | 71 |
| Caleb | 76 |
| Danielle | 87 |
| Evan | 85 |
| Faith | 92 |
| Gabe | 85 |
| Hannah | 99 |
| Ian | 73 |
| Jane | 68 |

**Transformation:**

You can use the following transformations to calculate the average (mean), minimum, and maximum scores:

| Transformation Name | New formula |
|---|---|
| **Parameter: Formula type** | Single row formula |
| **Parameter: Formula** | AVERAGE(Score) |
| **Parameter: New column name** | 'avgScore' |

| Transformation Name | New formula |
|---|---|
| **Parameter: Formula type** | Single row formula |
| **Parameter: Formula** | MIN(Score) |
| **Parameter: New column name** | 'minScore' |

| Transformation Name | New formula |
|---|---|
| **Parameter: Formula type** | Single row formula |
| **Parameter: Formula** | MAX(Score) |
| **Parameter: New column name** | 'maxScore' |

To apply statistical functions to your data, you can use the VAR and STDEV functions, which can be used as the basis for other statistical calculations.

| Transformation Name | New formula |
|---|---|
| **Parameter: Formula type** | Single row formula |
| **Parameter: Formula** | VAR(Score) |
| **Parameter: New column name** | var_Score |

| Transformation Name | New formula |
|---|---|
| **Parameter: Formula type** | Single row formula |
| **Parameter: Formula** | STDEV(Score) |
| **Parameter: New column name** | stdev_Score |

For each score, you can now calculate the variation of each one from the average, using the following:

| Transformation Name | New formula |
|---|---|
| **Parameter: Formula type** | Single row formula |

| | |
|---|---|
| **Parameter: Formula** | `((Score - avg_Score) / stdev_Score)` |
| **Parameter: New column name** | `'stDevs'` |

Now, you want to apply grades based on a formula:

| Grade | standard deviations from avg (stDevs) |
|---|---|
| A | stDevs > 1 |
| B | stDevs > 0.5 |
| C | -1 <= stDevs <= 0.5 |
| D | stDevs < -1 |
| F | stDevs < -2 |

You can build the following transformation using the `IF` function to calculate grades.

| | |
|---|---|
| **Transformation Name** | `New formula` |
| **Parameter: Formula type** | `Single row formula` |
| **Parameter: Formula** | `IF((stDevs > 1),'A',IF((stDevs < -2),'F',IF((stDevs < -1),'D',IF` `((stDevs > 0.5),'B','C'))))` |

To clean up the content, you might want to apply some formatting to the score columns. The following reformats the `stdev_Score` and `stDevs` columns to display two decimal places:

| | |
|---|---|
| **Transformation Name** | `Edit column with formula` |
| **Parameter: Columns** | `stdev_Score` |
| **Parameter: Formula** | `NUMFORMAT(stdev_Score, '##.00')` |

| | |
|---|---|
| **Transformation Name** | `Edit column with formula` |
| **Parameter: Columns** | `stDevs` |
| **Parameter: Formula** | `NUMFORMAT(stDevs, '##.00')` |

| | |
|---|---|
| **Transformation Name** | `New formula` |
| **Parameter: Formula type** | `Single row formula` |
| **Parameter: Formula** | `MODE(Score)` |
| **Parameter: New column name** | `'modeScore'` |

**Results:**

| Student | Score | modeScore | avgScore | minScore | maxScore | var_Score | stdev_Score | stDevs | Grade |
|---|---|---|---|---|---|---|---|---|---|
| Anna | 84 | 85 | 82 | 68 | 99 | 87.000000000 00001 | 9.33 | 0.21 | C |

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| Ben | 71 | 85 | 82 | 68 | 99 | 87.000000000 00001 | 9.33 | -1.18 | D |
| Caleb | 76 | 85 | 82 | 68 | 99 | 87.000000000 00001 | 9.33 | -0.64 | C |
| Danielle | 87 | 85 | 82 | 68 | 99 | 87.000000000 00001 | 9.33 | 0.54 | B |
| Evan | 85 | 85 | 82 | 68 | 99 | 87.000000000 00001 | 9.33 | 0.32 | C |
| Faith | 92 | 85 | 82 | 68 | 99 | 87.000000000 00001 | 9.33 | 1.07 | A |
| Gabe | 85 | 85 | 82 | 68 | 99 | 87.000000000 00001 | 9.33 | 0.32 | C |
| Hannah | 99 | 85 | 82 | 68 | 99 | 87.000000000 00001 | 9.33 | 1.82 | A |
| Ian | 73 | 85 | 82 | 68 | 99 | 87.000000000 00001 | 9.33 | -0.96 | C |
| Jane | 68 | 85 | 82 | 68 | 99 | 87.000000000 00001 | 9.33 | -1.50 | D |

# MAXIF Function

**Contents:**

Generates the maximum value of rows in each group that meet a specific condition. Inputs can be Integer, Decimal, or Datetime.

> **NOTE:** When added to a transformation, this function is applied to the current sample. If you change your sample or run the job, the computed values for this function are updated. Transformations that change the number of rows in subsequent recipe steps do not affect the values computed for this step.

To calculate the maximum value of rows without conditionals, use the `MAX` function. See *MAX Function*.

**Wrangle vs. SQL:** This function is part of Wrangle , a proprietary data transformation language. Wrangle is not SQL. For more information, see *Wrangle Language*.

## Basic Usage

```
maxif(testScores, testCount &gt;= 3)
```

**Output:** Returns the maximum of the `testScores` column when the `testCount` is greater than or equal to 3.

## Syntax and Arguments

```
maxif(col_ref, test_expression) [group:group_col_ref] [limit:limit_count]
```

| Argument | Required? | Data Type | Description |
|---|---|---|---|
| col_ref | Y | string | Reference to the column you wish to evaluate. |
| test_expression | Y | string | Expression that is evaluated. Must resolve to `true` or `false` |

For more information on syntax standards, see *Language Documentation Syntax Notes*.

For more information on the `group` and `limit` parameters, see *Pivot Transform*.

**col_ref**

Name of the column whose values you wish to use in the calculation. Inputs must be Integer, Decimal, or Datetime values.

> **NOTE:** If the input is in Datetime type, the output is in unixtime format. You can wrap these outputs in the DATEFORMAT function to generate the results in the appropriate Datetime format. See *DATEFORMAT Function*.

## Usage Notes:

| Required? | Data Type | Example Value |
|-----------|-----------|---------------|
| Yes | String that corresponds to the name of the column | `myValues` |

### test_expression

This parameter contains the expression to evaluate. This expression must resolve to a Boolean (`true` or `false`) value.

## Usage Notes:

| Required? | Data Type | Example Value |
|-----------|-----------|---------------|
| Yes | String expression that evaluates to `true` or `false` | `(LastName == 'Mouse' && FirstName == 'Mickey')` |

## Examples

> **Tip:** For additional examples, see *Common Tasks*.

### Example - Conditional Calculation Functions

This example illustrates how to use the conditional calculation functions.

**Functions**:

| Item | Description |
|------|-------------|
| AVERAGEIF Function | Generates the average value of rows in each group that meet a specific condition. Generated value is of Decimal type. |
| MINIF Function | Generates the minimum value of rows in each group that meet a specific condition. Inputs can be Integer, Decimal, or Datetime. |
| MAXIF Function | Generates the maximum value of rows in each group that meet a specific condition. Inputs can be Integer, Decimal, or Datetime. |
| VARIF Function | Generates the variance of values by group in a column that meet a specific condition. |
| STDEVIF Function | Generates the standard deviation of values by group in a column that meet a specific condition. |

### Source:

Here is some example weather data:

| date | city | rain | temp | wind |
|------|------|------|------|------|
| 1/23/17 | Valleyville | 0.00 | 12.8 | 6.7 |
| 1/23/17 | Center Town | 0.31 | 9.4 | 5.3 |

| 1/23/17 | Magic Mountain | 0.00 | 0.0 | 7.3 |
|---------|----------------|------|------|-----|
| 1/24/17 | Valleyville | 0.25 | 17.2 | 3.3 |
| 1/24/17 | Center Town | 0.54 | 1.1 | 7.6 |
| 1/24/17 | Magic Mountain | 0.32 | 5.0 | 8.8 |
| 1/25/17 | Valleyville | 0.02 | 3.3 | 6.8 |
| 1/25/17 | Center Town | 0.83 | 3.3 | 5.1 |
| 1/25/17 | Magic Mountain | 0.59 | -1.7 | 6.4 |
| 1/26/17 | Valleyville | 1.08 | 15.0 | 4.2 |
| 1/26/17 | Center Town | 0.96 | 6.1 | 7.6 |
| 1/26/17 | Magic Mountain | 0.77 | -3.9 | 3.0 |
| 1/27/17 | Valleyville | 1.00 | 7.2 | 2.8 |
| 1/27/17 | Center Town | 1.32 | 20.0 | 0.2 |
| 1/27/17 | Magic Mountain | 0.77 | 5.6 | 5.2 |
| 1/28/17 | Valleyville | 0.12 | -6.1 | 5.1 |
| 1/28/17 | Center Town | 0.14 | 5.0 | 4.9 |
| 1/28/17 | Magic Mountain | 1.50 | 1.1 | 0.4 |
| 1/29/17 | Valleyville | 0.36 | 13.3 | 7.3 |
| 1/29/17 | Center Town | 0.75 | 6.1 | 9.0 |
| 1/29/17 | Magic Mountain | 0.60 | 3.3 | 6.0 |

**Transformation:**

The following computes average temperature for rainy days by city:

| Transformation Name | `New formula` |
|---------------------|---------------|
| Parameter: Formula type | `Single row formula` |
| Parameter: Formula | `AVERAGEIF(temp, rain > 0)` |
| Parameter: Group rows by | `city` |
| Parameter: New column name | `'avgTempWRain'` |

The following computes maximum wind for sub-zero days by city:

| Transformation Name | `New formula` |
|---------------------|---------------|
| Parameter: Formula type | `Single row formula` |
| Parameter: Formula | `MAXIF(wind,temp < 0)` |
| Parameter: Group rows by | `city` |
| Parameter: New column name | `'maxWindSubZero'` |

This step calculates the minimum temp when the wind is less than 5 mph by city:

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | MINIF(temp,wind<5) |
| Parameter: Group rows by | city |
| Parameter: New column name | 'minTempWind5' |

This step computes the variance in temperature for rainy days by city:

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | VARIF(temp,rain >0) |
| Parameter: Group rows by | city |
| Parameter: New column name | 'varTempWRain' |

The following computes the standard deviation in rainfall for Center Town:

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | STDEVIF(rain,city=='Center Town') |
| Parameter: Group rows by | city |
| Parameter: New column name | 'stDevRainCT' |

You can use the following transforms to format the generated output. Note the `$col` placeholder value for the multi-column transforms:

| Transformation Name | Edit column with formula |
|---|---|
| Parameter: Columns | stDevRainCenterTown,maxWindSubZero |
| Parameter: Formula | numformat($col,'##.##') |

Since the following rely on data that has only one significant digit, you should format them differently:

| Transformation Name | Edit column with formula |
|---|---|
| Parameter: Columns | varTempWRain,avgTempWRain,minTempWind5 |
| Parameter: Formula | numformat($col,'##.#') |

**Results:**

| date | city | rain | temp | wind | avgTempWRain | maxWindSubZero | minTempWind5 | varTempWRain | stDevRain |
|---|---|---|---|---|---|---|---|---|---|
| 1/23/17 | Valley ville | 0.00 | 12.8 | 6.7 | 8.3 | 5.1 | 7.2 | 63.8 | 0.37 |

| Date | Location | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 1/23/17 | Center Town | 0.31 | 9.4 | 5.3 | 7.3 | | 5 | 32.6 | 0.37 |
| 1/23/17 | Magic Mountain | 0.00 | 0.0 | 7.3 | 1.6 | 6.43 | -3.9 | 12 | 0.37 |
| 1/24/17 | Valleyville | 0.25 | 17.2 | 3.3 | 8.3 | 5.1 | 7.2 | 63.8 | 0.37 |
| 1/24/17 | Center Town | 0.54 | 1.1 | 7.6 | 7.3 | | 5 | 32.6 | 0.37 |
| 1/24/17 | Magic Mountain | 0.32 | 5.0 | 8.8 | 1.6 | 6.43 | -3.9 | 12 | 0.37 |
| 1/25/17 | Valleyville | 0.02 | 3.3 | 6.8 | 8.3 | 5.1 | 7.2 | 63.8 | 0.37 |
| 1/25/17 | Center Town | 0.83 | 3.3 | 5.1 | 7.3 | | 5 | 32.6 | 0.37 |
| 1/25/17 | Magic Mountain | 0.59 | -1.7 | 6.4 | 1.6 | 6.43 | -3.9 | 12 | 0.37 |
| 1/26/17 | Valleyville | 1.08 | 15.0 | 4.2 | 8.3 | 5.1 | 7.2 | 63.8 | 0.37 |
| 1/26/17 | Center Town | 0.96 | 6.1 | 7.6 | 7.3 | | 5 | 32.6 | 0.37 |
| 1/26/17 | Magic Mountain | 0.77 | -3.9 | 3.0 | 1.6 | 6.43 | -3.9 | 12 | 0.37 |
| 1/27/17 | Valleyville | 1.00 | 7.2 | 2.8 | 8.3 | 5.1 | 7.2 | 63.8 | 0.37 |
| 1/27/17 | Center Town | 1.32 | 20.0 | 0.2 | 7.3 | | 5 | 32.6 | 0.37 |
| 1/27/17 | Magic Mountain | 0.77 | 5.6 | 5.2 | 1.6 | 6.43 | -3.9 | 12 | 0.37 |
| 1/28/17 | Valleyville | 0.12 | -6.1 | 5.1 | 8.3 | 5.1 | 7.2 | 63.8 | 0.37 |
| 1/28/17 | Center Town | 0.14 | 5.0 | 4.9 | 7.3 | | 5 | 32.6 | 0.37 |
| 1/28/17 | Magic Mountain | 1.50 | 1.1 | 0.4 | 1.6 | 6.43 | -3.9 | 12 | 0.37 |
| 1/29/17 | Valleyville | 0.36 | 13.3 | 7.3 | 8.3 | 5.1 | 7.2 | 63.8 | 0.37 |
| 1/29/17 | Center Town | 0.75 | 6.1 | 9.0 | 7.3 | | 5 | 32.6 | 0.37 |
| 1/29/17 | Magic Mountain | 0.60 | 3.3 | 6.0 | 1.6 | 6.43 | -3.9 | 12 | 0.37 |

# MEDIAN Function

Computes the median from all row values in a column or group. Input column can be of Integer or Decimal.

- If a row contains a missing or null value, it is not factored into the calculation. If the entire column contains no values, the function returns a null value.
- When used in a `pivot` transform, the function is computed for each instance of the value specified in the `group` parameter. See *Pivot Transform*.

**Wrangle vs. SQL:** This function is part of Wrangle , a proprietary data transformation language. Wrangle is not SQL. For more information, see *Wrangle Language*.

## Basic Usage

```
median(myRating)
```

**Output:** Returns the median of the values in the `myRating` column.

## Syntax and Arguments

```
median(function_col_ref) [group:group_col_ref] [limit:limit_count]
```

| Argument | Required? | Data Type | Description |
|---|---|---|---|
| function_col_ref | Y | string | Name of column to which to apply the function |

For more information on the `group` and `limit` parameters, see *Pivot Transform*.

For more information on syntax standards, see *Language Documentation Syntax Notes*.

### function_col_ref

Name of the column the values of which you want to calculate the median. Column must contain Integer or Decimal values.

- Literal values are not supported as inputs.
- Multiple columns and wildcards are not supported.

**Usage Notes:**

| Required? | Data Type | Example Value |
|---|---|---|
| Yes | String (column reference) | `myValues` |

## Examples

> **Tip:** For additional examples, see *Common Tasks*.

**Example - Percentile functions**

This example illustrates you to apply percentile functions.

**Functions**:

| Item | Description |
|---|---|
| MEDIAN Function | Computes the median from all row values in a column or group. Input column can be of Integer or Decimal. |
| PERCENTILE Function | Computes a specified percentile across all row values in a column or group. Input column can be of Integer or Decimal. |
| QUARTILE Function | Computes a specified quartile across all row values in a column or group. Input column can be of Integer or Decimal. |
| APPROXIMATEMEDIAN Function | Computes the approximate median from all row values in a column or group. Input column can be of Integer or Decimal. |
| APPROXIMATEPERCENTILE Function | Computes an approximation for a specified percentile across all row values in a column or group. Input column can be of Integer or Decimal. |
| APPROXIMATEQUARTILE Function | Computes an approximation for a specified quartile across all row values in a column or group. Input column can be of Integer or Decimal. |

### Source:

The following table lists each student's height in inches:

| Student | Height |
|---|---|
| 1 | 64 |
| 2 | 65 |
| 3 | 63 |
| 4 | 64 |
| 5 | 62 |
| 6 | 66 |
| 7 | 66 |
| 8 | 65 |
| 9 | 69 |
| 10 | 66 |
| 11 | 73 |
| 12 | 69 |
| 13 | 69 |
| 14 | 61 |
| 15 | 64 |
| 16 | 61 |
| 17 | 71 |
| 18 | 67 |
| 19 | 73 |
| 20 | 66 |

### Transformation:

Use the following transformations to calculate the median height in inches, a specified percentile and the first quartile.

- The first function uses a precise algorithm which can be slow to execute across large datasets.
- The second function uses an appropriate approximation algorithm, which is much faster to execute across large datasets.
  - These approximate functions can use an error boundary parameter, which is set to `0.4` (0.4%) across all functions.

**Median:** This transformation calculates the median value, which corresponds to the 50th percentile.

| Transformation Name | New formula |
| --- | --- |
| **Parameter: Formula type** | `Single row formula` |
| **Parameter: Formula** | `median(heightIn)` |
| **Parameter: New column name** | `'medianIn'` |

| Transformation Name | New formula |
| --- | --- |
| **Parameter: Formula type** | `Single row formula` |
| **Parameter: Formula** | `approximatemedian(heightIn, 0.4)` |
| **Parameter: New column name** | `'approxMedianIn'` |

**Percentile:** This transformation calculates the 68th percentile.

| Transformation Name | New formula |
| --- | --- |
| **Parameter: Formula type** | `Single row formula` |
| **Parameter: Formula** | `percentile(heightIn, 68, linear)` |
| **Parameter: New column name** | `'percentile68In'` |

| Transformation Name | New formula |
| --- | --- |
| **Parameter: Formula type** | `Single row formula` |
| **Parameter: Formula** | `approximatepercentile(heightIn, 68, 0.4)` |
| **Parameter: New column name** | `'approxPercentile68In'` |

**Quartile:** This transformation calculates the first quartile, which corresponds to the 25th percentile.

| Transformation Name | New formula |
| --- | --- |
| **Parameter: Formula type** | `Single row formula` |
| **Parameter: Formula** | `quartile(heightIn, 1, linear)` |
| **Parameter: New column name** | `'percentile25In'` |

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | approximatequartile(heightIn, 1, 0.4) |
| Parameter: New column name | 'approxPercentile25In' |

**Results:**

| studentId | heightIn | approxPercentile25In | percentile25In | approxPercentile68In | percentile68In | approxMedianIn | |
|---|---|---|---|---|---|---|---|
| 1 | 64 | 64 | 64 | 67.1 | 66.92 | 66 | 6 |
| 2 | 65 | 64 | 64 | 67.1 | 66.92 | 66 | 6 |
| 3 | 63 | 64 | 64 | 67.1 | 66.92 | 66 | 6 |
| 4 | 64 | 64 | 64 | 67.1 | 66.92 | 66 | 6 |
| 5 | 62 | 64 | 64 | 67.1 | 66.92 | 66 | 6 |
| 6 | 66 | 64 | 64 | 67.1 | 66.92 | 66 | 6 |
| 7 | 66 | 64 | 64 | 67.1 | 66.92 | 66 | 6 |
| 8 | 65 | 64 | 64 | 67.1 | 66.92 | 66 | 6 |
| 9 | 69 | 64 | 64 | 67.1 | 66.92 | 66 | 6 |
| 10 | 66 | 64 | 64 | 67.1 | 66.92 | 66 | 6 |
| 11 | 73 | 64 | 64 | 67.1 | 66.92 | 66 | 6 |
| 12 | 69 | 64 | 64 | 67.1 | 66.92 | 66 | 6 |
| 13 | 69 | 64 | 64 | 67.1 | 66.92 | 66 | 6 |
| 14 | 61 | 64 | 64 | 67.1 | 66.92 | 66 | 6 |
| 15 | 64 | 64 | 64 | 67.1 | 66.92 | 66 | 6 |
| 16 | 61 | 64 | 64 | 67.1 | 66.92 | 66 | 6 |
| 17 | 71 | 64 | 64 | 67.1 | 66.92 | 66 | 6 |
| 18 | 67 | 64 | 64 | 67.1 | 66.92 | 66 | 6 |
| 19 | 73 | 64 | 64 | 67.1 | 66.92 | 66 | 6 |
| 20 | 66 | 64 | 64 | 67.1 | 66.92 | 66 | 6 |

# MIN Function

Computes the minimum value found in all row values in a column. Input column can be of Integer, Decimal or Datetime.

- If a row contains a missing or null value, it is not factored into the calculation. If no numeric values are found in the source column, the function returns a null value.
- When used in a `pivot` transform, the function is computed for each instance of the value specified in the `group` parameter. See *Pivot Transform*.

For a version of this function computed over a rolling window of rows, see *ROLLINGMIN Function*.

Datetime inputs to this function return Unixtime values.

- These values can be wrapped in a DATEFORMAT function. See *DATEFORMAT Function*.
- For a date-native version of this function, see *MINDATE Function*.

**Wrangle vs. SQL:** This function is part of Wrangle , a proprietary data transformation language. Wrangle is not SQL. For more information, see *Wrangle Language*.

## Basic Usage

```
min(myRating)
```

**Output:** Returns the minimum value from the `myRating` column.

## Syntax and Arguments

```
min(function_col_ref) [group:group_col_ref] [limit:limit_count]
```

| Argument | Required? | Data Type | Description |
|---|---|---|---|
| function_col_ref | Y | string | Name of column to which to apply the function |

For more information on the `group` and `limit` parameters, see *Pivot Transform*.

For more information on syntax standards, see *Language Documentation Syntax Notes*.

**function_col_ref**

Name of the column the values of which you want to calculate the minimum. Inputs must be Integer, Decimal, or Datetime values.

> **NOTE:** If the input is in Datetime type, the output is in unixtime format. You can wrap these outputs in the DATEFORMAT function to generate the results in the appropriate Datetime format. See *DATEFORMAT Function*.

- Literal values are not supported as inputs.
- Multiple columns and wildcards are not supported.

**Usage Notes:**

| Required? | Data Type | Example Value |
|---|---|---|
| Yes | String (column reference) | `myValues` |

## Examples

> **Tip:** For additional examples, see *Common Tasks.*

This example illustrates how you can apply statistical functions to your dataset. Calculations include average (mean), max, min, standard deviation, and variance.

**Functions:**

| Item | Description |
|---|---|
| AVERAGE Function | Computes the average (mean) from all row values in a column or group. Input column can be of Integer or Decimal. |
| MIN Function | Computes the minimum value found in all row values in a column. Input column can be of Integer, Decimal or Datetime. |
| MAX Function | Computes the maximum value found in all row values in a column. Inputs can be Integer, Decimal, or Datetime. |
| VAR Function | Computes the variance among all values in a column. Input column can be of Integer or Decimal. If no numeric values are detected in the input column, the function returns 0. |
| STDEV Function | Computes the standard deviation across all column values of Integer or Decimal type. |
| NUMFORMAT Function | Formats a numeric set of values according to the specified number formatting. Source values can be a literal numeric value, a function returning a numeric value, or reference to a column containing an Integer or Decimal values. |
| MODE Function | Computes the mode (most frequent value) from all row values in a column, according to their grouping. Input column can be of Integer, Decimal, or Datetime type. |

**Source:**

Students took a test and recorded the following scores. You want to perform some statistical analysis on them:

| Student | Score |
|---|---|
| Anna | 84 |
| Ben | 71 |
| Caleb | 76 |
| Danielle | 87 |
| Evan | 85 |
| Faith | 92 |
| Gabe | 85 |
| Hannah | 99 |
| Ian | 73 |
| Jane | 68 |

**Transformation:**

You can use the following transformations to calculate the average (mean), minimum, and maximum scores:

| Transformation Name | New formula |
|---|---|
| **Parameter: Formula type** | Single row formula |
| **Parameter: Formula** | AVERAGE(Score) |
| **Parameter: New column name** | 'avgScore' |

| Transformation Name | New formula |
|---|---|
| **Parameter: Formula type** | Single row formula |
| **Parameter: Formula** | MIN(Score) |
| **Parameter: New column name** | 'minScore' |

| Transformation Name | New formula |
|---|---|
| **Parameter: Formula type** | Single row formula |
| **Parameter: Formula** | MAX(Score) |
| **Parameter: New column name** | 'maxScore' |

To apply statistical functions to your data, you can use the VAR and STDEV functions, which can be used as the basis for other statistical calculations.

| Transformation Name | New formula |
|---|---|
| **Parameter: Formula type** | Single row formula |
| **Parameter: Formula** | VAR(Score) |
| **Parameter: New column name** | var_Score |

| Transformation Name | New formula |
|---|---|
| **Parameter: Formula type** | Single row formula |
| **Parameter: Formula** | STDEV(Score) |
| **Parameter: New column name** | stdev_Score |

For each score, you can now calculate the variation of each one from the average, using the following:

| Transformation Name | New formula |
|---|---|
| **Parameter: Formula type** | Single row formula |

| Parameter: Formula | `((Score - avg_Score) / stdev_Score)` |
|---|---|
| Parameter: New column name | `'stDevs'` |

Now, you want to apply grades based on a formula:

| Grade | standard deviations from avg (stDevs) |
|---|---|
| A | stDevs > 1 |
| B | stDevs > 0.5 |
| C | -1 <= stDevs <= 0.5 |
| D | stDevs < -1 |
| F | stDevs < -2 |

You can build the following transformation using the `IF` function to calculate grades.

| Transformation Name | `New formula` |
|---|---|
| Parameter: Formula type | `Single row formula` |
| Parameter: Formula | `IF((stDevs > 1),'A',IF((stDevs < -2),'F',IF((stDevs < -1),'D',IF ((stDevs > 0.5),'B','C'))))` |

To clean up the content, you might want to apply some formatting to the score columns. The following reformats the `stdev_Score` and `stDevs` columns to display two decimal places:

| Transformation Name | `Edit column with formula` |
|---|---|
| Parameter: Columns | `stdev_Score` |
| Parameter: Formula | `NUMFORMAT(stdev_Score, '##.00')` |

| Transformation Name | `Edit column with formula` |
|---|---|
| Parameter: Columns | `stDevs` |
| Parameter: Formula | `NUMFORMAT(stDevs, '##.00')` |

| Transformation Name | `New formula` |
|---|---|
| Parameter: Formula type | `Single row formula` |
| Parameter: Formula | `MODE(Score)` |
| Parameter: New column name | `'modeScore'` |

**Results:**

| Student | Score | modeScore | avgScore | minScore | maxScore | var_Score | stdev_Score | stDevs | Grade |
|---|---|---|---|---|---|---|---|---|---|
| Anna | 84 | 85 | 82 | 68 | 99 | 87.000000000 00001 | 9.33 | 0.21 | C |

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| Ben | 71 | 85 | 82 | 68 | 99 | 87.000000000 00001 | 9.33 | -1.18 | D |
| Caleb | 76 | 85 | 82 | 68 | 99 | 87.000000000 00001 | 9.33 | -0.64 | C |
| Danielle | 87 | 85 | 82 | 68 | 99 | 87.000000000 00001 | 9.33 | 0.54 | B |
| Evan | 85 | 85 | 82 | 68 | 99 | 87.000000000 00001 | 9.33 | 0.32 | C |
| Faith | 92 | 85 | 82 | 68 | 99 | 87.000000000 00001 | 9.33 | 1.07 | A |
| Gabe | 85 | 85 | 82 | 68 | 99 | 87.000000000 00001 | 9.33 | 0.32 | C |
| Hannah | 99 | 85 | 82 | 68 | 99 | 87.000000000 00001 | 9.33 | 1.82 | A |
| Ian | 73 | 85 | 82 | 68 | 99 | 87.000000000 00001 | 9.33 | -0.96 | C |
| Jane | 68 | 85 | 82 | 68 | 99 | 87.000000000 00001 | 9.33 | -1.50 | D |

# MINIF Function

**Contents:**

Generates the minimum value of rows in each group that meet a specific condition. Inputs can be Integer, Decimal, or Datetime.

> **NOTE:** When added to a transformation, this function is applied to the current sample. If you change your sample or run the job, the computed values for this function are updated. Transformations that change the number of rows in subsequent recipe steps do not affect the values computed for this step.

To calculate the minimum value of rows without conditionals, use the `MIN` function. See *MIN Function*.

**Wrangle vs. SQL:** This function is part of Wrangle , a proprietary data transformation language. Wrangle is not SQL. For more information, see *Wrangle Language*.

## Basic Usage

```
minif(testScores, testCount &gt;= 3)
```

**Output:** Returns the minimum of the `testScores` column when the `testCount` is greater than or equal to 3.

## Syntax and Arguments

```
minif(col_ref, test_expression) [group:group_col_ref] [limit:limit_count]
```

| Argument | Required? | Data Type | Description |
|---|---|---|---|
| col_ref | Y | string | Reference to the column you wish to evaluate. |
| test_expression | Y | string | Expression that is evaluated. Must resolve to `true` or `false` |

For more information on syntax standards, see *Language Documentation Syntax Notes*.

For more information on the `group` and `limit` parameters, see *Pivot Transform*.

**col_ref**

Name of the column whose values you wish to use in the calculation. Inputs must be Integer, Decimal, or Datetime values.

> **NOTE:** If the input is in Datetime type, the output is in unixtime format. You can wrap these outputs in the DATEFORMAT function to generate the results in the appropriate Datetime format. See *DATEFORMAT Function*.

**Usage Notes:**

| Required? | Data Type | Example Value |
|---|---|---|
| Yes | String that corresponds to the name of the column | `myValues` |

### test_expression

This parameter contains the expression to evaluate. This expression must resolve to a Boolean (`true` or `false`) value.

**Usage Notes:**

| Required? | Data Type | Example Value |
|---|---|---|
| Yes | String expression that evaluates to `true` or `false` | `(LastName == 'Mouse' && FirstName == 'Mickey')` |

## Examples

> **Tip:** For additional examples, see *Common Tasks*.

### Example - Conditional Calculation Functions

This example illustrates how to use the conditional calculation functions.

**Functions**:

| Item | Description |
|---|---|
| AVERAGEIF Function | Generates the average value of rows in each group that meet a specific condition. Generated value is of Decimal type. |
| MINIF Function | Generates the minimum value of rows in each group that meet a specific condition. Inputs can be Integer, Decimal, or Datetime. |
| MAXIF Function | Generates the maximum value of rows in each group that meet a specific condition. Inputs can be Integer, Decimal, or Datetime. |
| VARIF Function | Generates the variance of values by group in a column that meet a specific condition. |
| STDEVIF Function | Generates the standard deviation of values by group in a column that meet a specific condition. |

**Source:**

Here is some example weather data:

| date | city | rain | temp | wind |
|---|---|---|---|---|

| 1/23/17 | Valleyville | 0.00 | 12.8 | 6.7 |
|---|---|---|---|---|
| 1/23/17 | Center Town | 0.31 | 9.4 | 5.3 |
| 1/23/17 | Magic Mountain | 0.00 | 0.0 | 7.3 |
| 1/24/17 | Valleyville | 0.25 | 17.2 | 3.3 |
| 1/24/17 | Center Town | 0.54 | 1.1 | 7.6 |
| 1/24/17 | Magic Mountain | 0.32 | 5.0 | 8.8 |
| 1/25/17 | Valleyville | 0.02 | 3.3 | 6.8 |
| 1/25/17 | Center Town | 0.83 | 3.3 | 5.1 |
| 1/25/17 | Magic Mountain | 0.59 | -1.7 | 6.4 |
| 1/26/17 | Valleyville | 1.08 | 15.0 | 4.2 |
| 1/26/17 | Center Town | 0.96 | 6.1 | 7.6 |
| 1/26/17 | Magic Mountain | 0.77 | -3.9 | 3.0 |
| 1/27/17 | Valleyville | 1.00 | 7.2 | 2.8 |
| 1/27/17 | Center Town | 1.32 | 20.0 | 0.2 |
| 1/27/17 | Magic Mountain | 0.77 | 5.6 | 5.2 |
| 1/28/17 | Valleyville | 0.12 | -6.1 | 5.1 |
| 1/28/17 | Center Town | 0.14 | 5.0 | 4.9 |
| 1/28/17 | Magic Mountain | 1.50 | 1.1 | 0.4 |
| 1/29/17 | Valleyville | 0.36 | 13.3 | 7.3 |
| 1/29/17 | Center Town | 0.75 | 6.1 | 9.0 |
| 1/29/17 | Magic Mountain | 0.60 | 3.3 | 6.0 |

**Transformation:**

The following computes average temperature for rainy days by city:

| Transformation Name | New formula |
|---|---|
| **Parameter: Formula type** | Single row formula |
| **Parameter: Formula** | AVERAGEIF(temp, rain > 0) |
| **Parameter: Group rows by** | city |
| **Parameter: New column name** | 'avgTempWRain' |

The following computes maximum wind for sub-zero days by city:

| Transformation Name | New formula |
|---|---|
| **Parameter: Formula type** | Single row formula |
| **Parameter: Formula** | MAXIF(wind,temp < 0) |
| **Parameter: Group rows by** | city |
| **Parameter: New column name** | 'maxWindSubZero' |

This step calculates the minimum temp when the wind is less than 5 mph by city:

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | MINIF(temp,wind<5) |
| Parameter: Group rows by | city |
| Parameter: New column name | 'minTempWind5' |

This step computes the variance in temperature for rainy days by city:

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | VARIF(temp,rain >0) |
| Parameter: Group rows by | city |
| Parameter: New column name | 'varTempWRain' |

The following computes the standard deviation in rainfall for Center Town:

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | STDEVIF(rain,city=='Center Town') |
| Parameter: Group rows by | city |
| Parameter: New column name | 'stDevRainCT' |

You can use the following transforms to format the generated output. Note the $col placeholder value for the multi-column transforms:

| Transformation Name | Edit column with formula |
|---|---|
| Parameter: Columns | stDevRainCenterTown,maxWindSubZero |
| Parameter: Formula | numformat($col,'##.##') |

Since the following rely on data that has only one significant digit, you should format them differently:

| Transformation Name | Edit column with formula |
|---|---|
| Parameter: Columns | varTempWRain,avgTempWRain,minTempWind5 |
| Parameter: Formula | numformat($col,'##.#') |

**Results:**

| date | city | rain | temp | wind | avgTempWRain | maxWindSubZero | minTempWind5 | varTempWRain | stDevRain |
|---|---|---|---|---|---|---|---|---|---|
| 1/23/17 | Valley ville | 0.00 | 12.8 | 6.7 | 8.3 | 5.1 | 7.2 | 63.8 | 0.37 |
| 1/23/17 | Center Town | 0.31 | 9.4 | 5.3 | 7.3 | | 5 | 32.6 | 0.37 |
| 1/23/17 | Magic Mountain | 0.00 | 0.0 | 7.3 | 1.6 | 6.43 | -3.9 | 12 | 0.37 |
| 1/24/17 | Valley ville | 0.25 | 17.2 | 3.3 | 8.3 | 5.1 | 7.2 | 63.8 | 0.37 |
| 1/24/17 | Center Town | 0.54 | 1.1 | 7.6 | 7.3 | | 5 | 32.6 | 0.37 |
| 1/24/17 | Magic Mountain | 0.32 | 5.0 | 8.8 | 1.6 | 6.43 | -3.9 | 12 | 0.37 |
| 1/25/17 | Valley ville | 0.02 | 3.3 | 6.8 | 8.3 | 5.1 | 7.2 | 63.8 | 0.37 |
| 1/25/17 | Center Town | 0.83 | 3.3 | 5.1 | 7.3 | | 5 | 32.6 | 0.37 |
| 1/25/17 | Magic Mountain | 0.59 | -1.7 | 6.4 | 1.6 | 6.43 | -3.9 | 12 | 0.37 |
| 1/26/17 | Valley ville | 1.08 | 15.0 | 4.2 | 8.3 | 5.1 | 7.2 | 63.8 | 0.37 |
| 1/26/17 | Center Town | 0.96 | 6.1 | 7.6 | 7.3 | | 5 | 32.6 | 0.37 |
| 1/26/17 | Magic Mountain | 0.77 | -3.9 | 3.0 | 1.6 | 6.43 | -3.9 | 12 | 0.37 |
| 1/27/17 | Valley ville | 1.00 | 7.2 | 2.8 | 8.3 | 5.1 | 7.2 | 63.8 | 0.37 |
| 1/27/17 | Center Town | 1.32 | 20.0 | 0.2 | 7.3 | | 5 | 32.6 | 0.37 |
| 1/27/17 | Magic Mountain | 0.77 | 5.6 | 5.2 | 1.6 | 6.43 | -3.9 | 12 | 0.37 |
| 1/28/17 | Valley ville | 0.12 | -6.1 | 5.1 | 8.3 | 5.1 | 7.2 | 63.8 | 0.37 |
| 1/28/17 | Center Town | 0.14 | 5.0 | 4.9 | 7.3 | | 5 | 32.6 | 0.37 |
| 1/28/17 | Magic Mountain | 1.50 | 1.1 | 0.4 | 1.6 | 6.43 | -3.9 | 12 | 0.37 |
| 1/29/17 | Valley ville | 0.36 | 13.3 | 7.3 | 8.3 | 5.1 | 7.2 | 63.8 | 0.37 |
| 1/29/17 | Center Town | 0.75 | 6.1 | 9.0 | 7.3 | | 5 | 32.6 | 0.37 |
| 1/29/17 | Magic Mountain | 0.60 | 3.3 | 6.0 | 1.6 | 6.43 | -3.9 | 12 | 0.37 |

# MODE Function

Computes the mode (most frequent value) from all row values in a column, according to their grouping. Input column can be of Integer, Decimal, or Datetime type.

- If a row contains a missing or null value, it is not factored into the calculation. If the entire column contains no values, the function returns a null value.
- If there is a tie in which the most occurrences of a value is shared between values, then the lowest value of the evaluated set is returned.
- When used in a `pivot` transform, the function is computed for each instance of the value specified in the `group` parameter. See *Pivot Transform*.

For a version of this function computed over a rolling window of rows, see *ROLLINGMODE Function*.

Datetime inputs to this function return Unixtime values.

- These values can be wrapped in a DATEFORMAT function. See *DATEFORMAT Function*.
- For a date-native version of this function, see *MODEDATE Function*.

**Wrangle vs. SQL:** This function is part of Wrangle , a proprietary data transformation language. Wrangle is not SQL. For more information, see *Wrangle Language*.

## Basic Usage

```
mode(count_visits)
```

**Output:** Returns the mode of the values in the `count_visits` column.

## Syntax and Arguments

```
mode(function_col_ref) [group:group_col_ref] [limit:limit_count]
```

| Argument | Required? | Data Type | Description |
|---|---|---|---|
| function_col_ref | Y | string | Name of column to which to apply the function |

For more information on the `group` and `limit` parameters, see *Pivot Transform*.

For more information on syntax standards, see *Language Documentation Syntax Notes*.

**function_col_ref**

Name of the column the values of which you want to calculate the function. Column must contain Integer, Decimal, or Datetime values.

> **NOTE:** If the input is in Datetime type, the output is in unixtime format. You can wrap these outputs in the DATEFORMAT function to generate the results in the appropriate Datetime format. See *DATEFORMAT Function*.

- Literal values are not supported as inputs.
- Multiple columns and wildcards are not supported.

**Usage Notes:**

| Required? | Data Type | Example Value |
|---|---|---|
| Yes | String (column reference) | `myValues` |

## Examples

> **Tip:** For additional examples, see *Common Tasks*.

### Example - Statistics on Test Scores

This example illustrates how you can apply statistical functions to your dataset. Calculations include average (mean), max, min, standard deviation, and variance.

**Functions:**

| Item | Description |
|---|---|
| AVERAGE Function | Computes the average (mean) from all row values in a column or group. Input column can be of Integer or Decimal. |
| MIN Function | Computes the minimum value found in all row values in a column. Input column can be of Integer, Decimal or Datetime. |
| MAX Function | Computes the maximum value found in all row values in a column. Inputs can be Integer, Decimal, or Datetime. |
| VAR Function | Computes the variance among all values in a column. Input column can be of Integer or Decimal. If no numeric values are detected in the input column, the function returns 0. |
| STDEV Function | Computes the standard deviation across all column values of Integer or Decimal type. |
| NUMFORMAT Function | Formats a numeric set of values according to the specified number formatting. Source values can be a literal numeric value, a function returning a numeric value, or reference to a column containing an Integer or Decimal values. |
| MODE Function | Computes the mode (most frequent value) from all row values in a column, according to their grouping. Input column can be of Integer, Decimal, or Datetime type. |

**Source:**

Students took a test and recorded the following scores. You want to perform some statistical analysis on them:

| Student | Score |
|---|---|
| Anna | 84 |
| Ben | 71 |
| Caleb | 76 |
| Danielle | 87 |
| Evan | 85 |
| Faith | 92 |
| Gabe | 85 |
| Hannah | 99 |

| | |
|---|---|
| Ian | 73 |
| Jane | 68 |

**Transformation:**

You can use the following transformations to calculate the average (mean), minimum, and maximum scores:

| Transformation Name | New formula |
|---|---|
| **Parameter: Formula type** | Single row formula |
| **Parameter: Formula** | AVERAGE(Score) |
| **Parameter: New column name** | 'avgScore' |

| Transformation Name | New formula |
|---|---|
| **Parameter: Formula type** | Single row formula |
| **Parameter: Formula** | MIN(Score) |
| **Parameter: New column name** | 'minScore' |

| Transformation Name | New formula |
|---|---|
| **Parameter: Formula type** | Single row formula |
| **Parameter: Formula** | MAX(Score) |
| **Parameter: New column name** | 'maxScore' |

To apply statistical functions to your data, you can use the VAR and STDEV functions, which can be used as the basis for other statistical calculations.

| Transformation Name | New formula |
|---|---|
| **Parameter: Formula type** | Single row formula |
| **Parameter: Formula** | VAR(Score) |
| **Parameter: New column name** | var_Score |

| Transformation Name | New formula |
|---|---|
| **Parameter: Formula type** | Single row formula |
| **Parameter: Formula** | STDEV(Score) |
| **Parameter: New column name** | stdev_Score |

For each score, you can now calculate the variation of each one from the average, using the following:

| Transformation Name | New formula |
|---|---|

| Parameter: Formula type | Single row formula |
|---|---|
| Parameter: Formula | ((Score - avg_Score) / stdev_Score) |
| Parameter: New column name | 'stDevs' |

Now, you want to apply grades based on a formula:

| Grade | standard deviations from avg (stDevs) |
|---|---|
| A | stDevs > 1 |
| B | stDevs > 0.5 |
| C | -1 <= stDevs <= 0.5 |
| D | stDevs < -1 |
| F | stDevs < -2 |

You can build the following transformation using the IF function to calculate grades.

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | IF((stDevs > 1),'A',IF((stDevs < -2),'F',IF((stDevs < -1),'D',IF ((stDevs > 0.5),'B','C')))) |

To clean up the content, you might want to apply some formatting to the score columns. The following reformats the stdev_Score and stDevs columns to display two decimal places:

| Transformation Name | Edit column with formula |
|---|---|
| Parameter: Columns | stdev_Score |
| Parameter: Formula | NUMFORMAT(stdev_Score, '##.00') |

| Transformation Name | Edit column with formula |
|---|---|
| Parameter: Columns | stDevs |
| Parameter: Formula | NUMFORMAT(stDevs, '##.00') |

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | MODE(Score) |
| Parameter: New column name | 'modeScore' |

**Results:**

| Student | Score | modeScore | avgScore | minScore | maxScore | var_Score | stdev_Score | stDevs | Grade |
|---|---|---|---|---|---|---|---|---|---|

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| Anna | 84 | 85 | 82 | 68 | 99 | 87.00000000000001 | 9.33 | 0.21 | C |
| Ben | 71 | 85 | 82 | 68 | 99 | 87.00000000000001 | 9.33 | -1.18 | D |
| Caleb | 76 | 85 | 82 | 68 | 99 | 87.00000000000001 | 9.33 | -0.64 | C |
| Danielle | 87 | 85 | 82 | 68 | 99 | 87.00000000000001 | 9.33 | 0.54 | B |
| Evan | 85 | 85 | 82 | 68 | 99 | 87.00000000000001 | 9.33 | 0.32 | C |
| Faith | 92 | 85 | 82 | 68 | 99 | 87.00000000000001 | 9.33 | 1.07 | A |
| Gabe | 85 | 85 | 82 | 68 | 99 | 87.00000000000001 | 9.33 | 0.32 | C |
| Hannah | 99 | 85 | 82 | 68 | 99 | 87.00000000000001 | 9.33 | 1.82 | A |
| Ian | 73 | 85 | 82 | 68 | 99 | 87.00000000000001 | 9.33 | -0.96 | C |
| Jane | 68 | 85 | 82 | 68 | 99 | 87.00000000000001 | 9.33 | -1.50 | D |

# MODEIF Function

**Contents:**

- *Basic Usage*
- *Syntax and Arguments*
  - *function_col_ref*
  - *test_expression*
- *Examples*
  - *Example - MODEIF function*

---

Computes the mode (most frequent value) from all row values in a column, according to their grouping. Input column can be of Integer, Decimal, or Datetime type.

- If a row contains a missing or null value, it is not factored into the calculation. If the entire column contains no values, the function returns a null value.
- If there is a tie in which the most occurrences of a value is shared between values, then the lowest value of the evaluated set is returned.
- When used in a `pivot` transform, the function is computed for each instance of the value specified in the `group` parameter. See *Pivot Transform*.

For a non-conditional version of this function, see *MODE Function*.

For a version of this function computed over a rolling window of rows, see *ROLLINGMODE Function*.

**Wrangle vs. SQL:** This function is part of Wrangle , a proprietary data transformation language. Wrangle is not SQL. For more information, see *Wrangle Language*.

## Basic Usage

```
modeif(count_visits, health_status == &apos;sick&apos;)
```

**Output:** Returns the mode of the values in the `count_visits` column as long as `health_status` is set to `sick`.

## Syntax and Arguments

```
modeif(function_col_ref, test_expression) [group:group_col_ref] [limit:limit_count]
```

| Argument | Required? | Data Type | Description |
|---|---|---|---|
| function_col_ref | Y | string | Name of column to which to apply the function |
| test_expression | Y | string | Expression that is evaluated. Must resolve to `true` or `false` |

For more information on the `group` and `limit` parameters, see *Pivot Transform*.

For more information on syntax standards, see *Language Documentation Syntax Notes*.

**function_col_ref**

Name of the column the values of which you want to calculate the function. Column must contain Integer, Decimal, or Datetime values.

> **NOTE:** If the input is in Datetime type, the output is in unixtime format. You can wrap these outputs in the DATEFORMAT function to generate the results in the appropriate Datetime format. See *DATEFORMAT Function*.

- Literal values are not supported as inputs.
- Multiple columns and wildcards are not supported.

**Usage Notes:**

| Required? | Data Type | Example Value |
|---|---|---|
| Yes | String (column reference) | `myValues` |

**test_expression**

This parameter contains the expression to evaluate. This expression must resolve to a Boolean (`true` or `false`) value.

**Usage Notes:**

| Required? | Data Type | Example Value |
|---|---|---|
| Yes | String expression that evaluates to `true` or `false` | `(LastName == 'Mouse' && FirstName == 'Mickey')` |

## Examples

> **Tip:** For additional examples, see *Common Tasks*.

**Example - MODEIF function**

The following data contains a list of weekly orders for 2017 across two regions (`r01` and `r02`). You are interested in calculating the most common order count for the second half of the year, by region.

**Source:**

> **NOTE:** For simplicity, only the first few rows are displayed.

| Date | Region | OrderCount |
|---|---|---|
| 1/6/2017 | r01 | 78 |
| 1/6/2017 | r02 | 97 |
| 1/13/2017 | r01 | 92 |

| | | |
|---|---|---|
| 1/13/2017 | r02 | 90 |
| 1/20/2017 | r01 | 97 |
| 1/20/2017 | r02 | 84 |

**Transformation:**

To assist, you can first calculate the week number for each row:

| Transformation Name | New formula |
|---|---|
| **Parameter: Formula type** | Single row formula |
| **Parameter: Formula** | weeknum(Date) |
| **Parameter: New column name** | 'weekNumber' |

Then, you can use the following aggregation to determine the most common order value for each region during the second half of the year:

| Transformation Name | Pivot columns |
|---|---|
| **Parameter: Row labels** | Region |
| **Parameter: Values** | modeif(OrderCount, weekNumber > 26) |
| **Parameter: Max number of columns to create** | 50 |

**Results:**

| Region | modeif_OrderCount |
|---|---|
| r01 | 85 |
| r02 | 100 |

# PERCENTILE Function

**Contents:**

---

Computes a specified percentile across all row values in a column or group. Input column can be of Integer or Decimal.

- If a row contains a missing or null value, it is not factored into the calculation. If the entire column contains no values, the function returns a null value.
- When used in a `pivot` transform, the function is computed for each instance of the value specified in the `group` parameter. See *Pivot Transform*.

**Wrangle vs. SQL:** This function is part of Wrangle , a proprietary data transformation language. Wrangle is not SQL. For more information, see *Wrangle Language*.

## Basic Usage

```
percentile(myScores, 25, linear)
```

**Output:** Computes the value that is at the 25th percentile across all values in the `myScores` column.

## Syntax and Arguments

```
percentile(function_col_ref,num_percentile,interpolation_method) [group:group_col_ref]
[limit:limit_count]
```

| Argument | Required? | Data Type | Description |
|---|---|---|---|
| function_col_ref | Y | string | Name of column to which to apply the function |
| num_percentile | Y | integer | Integer value between 1-100 of the percentile to compute |
| interpolation_method | Y | enum | Method by which to interpolate values between two row values. See below. |

For more information on the `group` and `limit` parameters, see *Pivot Transform*.

For more information on syntax standards, see *Language Documentation Syntax Notes*.

**function_col_ref**

Name of the column the values of which you want to calculate the percentile. Column must contain Integer or Decimal values.

- Literal values are not supported as inputs.
- Multiple columns and wildcards are not supported.

**Usage Notes:**

| Required? | Data Type | Example Value |
|---|---|---|
| Yes | String (column reference) | `precipitationI`<br>`n` |

### num_percentile

Integer literal value indicating the percentile that you wish to compute. Input value must be between `1` and `100`, inclusive.

- Column or function references are not supported.
- Multiple columns and wildcards are not supported.

**Usage Notes:**

| Required? | Data Type | Example Value |
|---|---|---|
| Yes | Integer | `55` |

### interpolation_method

Method of interpolation between each discrete value. The list of support methods is the following:

| Interpolation method | Description |
|---|---|
| `linear` | Percentiles are calculated between two discrete values in a linear fashion. |
| `exclusive` | Excludes 0 and 1 from calculation of percentiles. |
| `inclusive` | Includes 0 and 1 from calculation of percentiles. |
| `lower` | Use the lower value when the computed value falls between two discrete values. |
| `upper` | Use the upper value when the computed value falls between two discrete values. |
| `midpoint` | Use the midpoint value when the computed value falls between two discrete values. |

**Usage Notes:**

| Required? | Data Type | Example Value |
|---|---|---|
| Yes | Enum | `linear` |

## Examples

> **Tip:** For additional examples, see *Common Tasks*.

### Example - Percentile functions

This example illustrates you to apply percentile functions.

**Functions**:

| Item | Description |
|---|---|
| MEDIAN Function | Computes the median from all row values in a column or group. Input column can be of Integer or Decimal. |
| PERCENTILE Function | Computes a specified percentile across all row values in a column or group. Input column can be of Integer or Decimal. |
| QUARTILE Function | Computes a specified quartile across all row values in a column or group. Input column can be of Integer or Decimal. |
| APPROXIMATEMEDIAN Function | Computes the approximate median from all row values in a column or group. Input column can be of Integer or Decimal. |
| APPROXIMATEPERCENTIL E Function | Computes an approximation for a specified percentile across all row values in a column or group. Input column can be of Integer or Decimal. |
| APPROXIMATEQUARTILE Function | Computes an approximation for a specified quartile across all row values in a column or group. Input column can be of Integer or Decimal. |

**Source:**

The following table lists each student's height in inches:

| Student | Height |
|---|---|
| 1 | 64 |
| 2 | 65 |
| 3 | 63 |
| 4 | 64 |
| 5 | 62 |
| 6 | 66 |
| 7 | 66 |
| 8 | 65 |
| 9 | 69 |
| 10 | 66 |
| 11 | 73 |
| 12 | 69 |
| 13 | 69 |
| 14 | 61 |
| 15 | 64 |
| 16 | 61 |
| 17 | 71 |
| 18 | 67 |
| 19 | 73 |
| 20 | 66 |

**Transformation:**

Use the following transformations to calculate the median height in inches, a specified percentile and the first quartile.

- The first function uses a precise algorithm which can be slow to execute across large datasets.

- The second function uses an appropriate approximation algorithm, which is much faster to execute across large datasets.
    - These approximate functions can use an error boundary parameter, which is set to `0.4` (0.4%) across all functions.

**Median:** This transformation calculates the median value, which corresponds to the 50th percentile.

| | |
|---|---|
| **Transformation Name** | `New formula` |
| **Parameter: Formula type** | `Single row formula` |
| **Parameter: Formula** | `median(heightIn)` |
| **Parameter: New column name** | `'medianIn'` |

| | |
|---|---|
| **Transformation Name** | `New formula` |
| **Parameter: Formula type** | `Single row formula` |
| **Parameter: Formula** | `approximatemedian(heightIn, 0.4)` |
| **Parameter: New column name** | `'approxMedianIn'` |

**Percentile:** This transformation calculates the 68th percentile.

| | |
|---|---|
| **Transformation Name** | `New formula` |
| **Parameter: Formula type** | `Single row formula` |
| **Parameter: Formula** | `percentile(heightIn, 68, linear)` |
| **Parameter: New column name** | `'percentile68In'` |

| | |
|---|---|
| **Transformation Name** | `New formula` |
| **Parameter: Formula type** | `Single row formula` |
| **Parameter: Formula** | `approximatepercentile(heightIn, 68, 0.4)` |
| **Parameter: New column name** | `'approxPercentile68In'` |

**Quartile:** This transformation calculates the first quartile, which corresponds to the 25th percentile.

| | |
|---|---|
| **Transformation Name** | `New formula` |
| **Parameter: Formula type** | `Single row formula` |
| **Parameter: Formula** | `quartile(heightIn, 1, linear)` |
| **Parameter: New column name** | `'percentile25In'` |

| | |
|---|---|
| **Transformation Name** | `New formula` |
| **Parameter: Formula type** | `Single row formula` |

| Parameter: Formula | approximatequartile(heightIn, 1, 0.4) |
|---|---|
| Parameter: New column name | 'approxPercentile25In' |

**Results:**

| studentId | heightIn | approxPercentile25In | percentile25In | approxPercentile68In | percentile68In | approxMedianIn | |
|---|---|---|---|---|---|---|---|
| 1 | 64 | 64 | 64 | 67.1 | 66.92 | 66 | 6 |
| 2 | 65 | 64 | 64 | 67.1 | 66.92 | 66 | 6 |
| 3 | 63 | 64 | 64 | 67.1 | 66.92 | 66 | 6 |
| 4 | 64 | 64 | 64 | 67.1 | 66.92 | 66 | 6 |
| 5 | 62 | 64 | 64 | 67.1 | 66.92 | 66 | 6 |
| 6 | 66 | 64 | 64 | 67.1 | 66.92 | 66 | 6 |
| 7 | 66 | 64 | 64 | 67.1 | 66.92 | 66 | 6 |
| 8 | 65 | 64 | 64 | 67.1 | 66.92 | 66 | 6 |
| 9 | 69 | 64 | 64 | 67.1 | 66.92 | 66 | 6 |
| 10 | 66 | 64 | 64 | 67.1 | 66.92 | 66 | 6 |
| 11 | 73 | 64 | 64 | 67.1 | 66.92 | 66 | 6 |
| 12 | 69 | 64 | 64 | 67.1 | 66.92 | 66 | 6 |
| 13 | 69 | 64 | 64 | 67.1 | 66.92 | 66 | 6 |
| 14 | 61 | 64 | 64 | 67.1 | 66.92 | 66 | 6 |
| 15 | 64 | 64 | 64 | 67.1 | 66.92 | 66 | 6 |
| 16 | 61 | 64 | 64 | 67.1 | 66.92 | 66 | 6 |
| 17 | 71 | 64 | 64 | 67.1 | 66.92 | 66 | 6 |
| 18 | 67 | 64 | 64 | 67.1 | 66.92 | 66 | 6 |
| 19 | 73 | 64 | 64 | 67.1 | 66.92 | 66 | 6 |
| 20 | 66 | 64 | 64 | 67.1 | 66.92 | 66 | 6 |

# QUARTILE Function

**Contents:**

---

Computes a specified quartile across all row values in a column or group. Input column can be of Integer or Decimal.

- If a row contains a missing or null value, it is not factored into the calculation. If the entire column contains no values, the function returns a null value.
- When used in a `pivot` transform, the function is computed for each instance of the value specified in the `group` parameter. See *Pivot Transform*.

Quartiles are computed as follows:

| Quartile | Description |
|----------|-------------|
| 0 | Minimum value |
| 1 | 25th percentile |
| 2 | Median value |
| 3 | 75th percentile and higher |

**Wrangle vs. SQL:** This function is part of Wrangle , a proprietary data transformation language. Wrangle is not SQL. For more information, see *Wrangle Language*.

## Basic Usage

```
<span>quartile</span>(myScores, 3, linear)
```

**Output:** Computes the value that is at the 3rd quartile across all values in the `myScores` column.

## Syntax and Arguments

```
quartile(function_col_ref,num_quartile,interpolation_method) [group:group_col_ref] [limit:
limit_count]
```

| Argument | Required? | Data Type | Description |
|----------|-----------|-----------|-------------|
| function_col_ref | Y | string | Name of column to which to apply the function |
| num_quartile | Y | integer | Integer value (0-3) of the quartile to compute |
| interpolation_method | Y | enum | Method by which to interpolate values between two row values. See below. |

For more information on the `group` and `limit` parameters, see *Pivot Transform*.

For more information on syntax standards, see *Language Documentation Syntax Notes*.

### function_col_ref

Name of the column the values of which you want to calculate the quartile. Column must contain Integer or Decimal values.

- Literal values are not supported as inputs.
- Multiple columns and wildcards are not supported.

**Usage Notes:**

| Required? | Data Type | Example Value |
|-----------|-----------|---------------|
| Yes | String (column reference) | `precipitationI`<br>`n` |

### num_quartile

Integer literal value indicating the quartile that you wish to compute. Input value must be between `0` and `3`, inclusive.

- Column or function references are not supported.
- Multiple columns and wildcards are not supported.

**Usage Notes:**

| Required? | Data Type | Example Value |
|-----------|-----------|---------------|
| Yes | Integer | 3 |

### interpolation_method

Method of interpolation between each discrete value. The list of support methods is the following:

| Interpolation method | Description |
|----------------------|-------------|
| `linear` | Quartiles are calculated between two discrete values in a linear fashion. |
| `exclusive` | Excludes 0 (0th percentile) and 1 (100th percentile) from calculation of quartiles. |
| `inclusive` | Includes 0 (0th percentile) and 1 (100th percentile) from calculation of quartiles. |
| `lower` | Use the lower value when the computed value falls between two discrete values. |
| `upper` | Use the upper value when the computed value falls between two discrete values. |
| `midpoint` | Use the midpoint value when the computed value falls between two discrete values. |

**Usage Notes:**

| Required? | Data Type | Example Value |
|-----------|-----------|---------------|
| Yes | Enum | `linear` |

# Examples

> **Tip:** For additional examples, see *Common Tasks.*

### Example - Percentile functions

This example illustrates you to apply percentile functions.

**Functions**:

| Item | Description |
|---|---|
| MEDIAN Function | Computes the median from all row values in a column or group. Input column can be of Integer or Decimal. |
| PERCENTILE Function | Computes a specified percentile across all row values in a column or group. Input column can be of Integer or Decimal. |
| QUARTILE Function | Computes a specified quartile across all row values in a column or group. Input column can be of Integer or Decimal. |
| APPROXIMATEMEDIAN Function | Computes the approximate median from all row values in a column or group. Input column can be of Integer or Decimal. |
| APPROXIMATEPERCENTILE Function | Computes an approximation for a specified percentile across all row values in a column or group. Input column can be of Integer or Decimal. |
| APPROXIMATEQUARTILE Function | Computes an approximation for a specified quartile across all row values in a column or group. Input column can be of Integer or Decimal. |

**Source:**

The following table lists each student's height in inches:

| Student | Height |
|---|---|
| 1 | 64 |
| 2 | 65 |
| 3 | 63 |
| 4 | 64 |
| 5 | 62 |
| 6 | 66 |
| 7 | 66 |
| 8 | 65 |
| 9 | 69 |
| 10 | 66 |
| 11 | 73 |
| 12 | 69 |
| 13 | 69 |
| 14 | 61 |
| 15 | 64 |
| 16 | 61 |

| | |
|---|---|
| 17 | 71 |
| 18 | 67 |
| 19 | 73 |
| 20 | 66 |

**Transformation:**

Use the following transformations to calculate the median height in inches, a specified percentile and the first quartile.

- The first function uses a precise algorithm which can be slow to execute across large datasets.
- The second function uses an appropriate approximation algorithm, which is much faster to execute across large datasets.
    - These approximate functions can use an error boundary parameter, which is set to `0.4` (0.4%) across all functions.

**Median:** This transformation calculates the median value, which corresponds to the 50th percentile.

| | |
|---|---|
| **Transformation Name** | `New formula` |
| **Parameter: Formula type** | `Single row formula` |
| **Parameter: Formula** | `median(heightIn)` |
| **Parameter: New column name** | `'medianIn'` |

| | |
|---|---|
| **Transformation Name** | `New formula` |
| **Parameter: Formula type** | `Single row formula` |
| **Parameter: Formula** | `approximatemedian(heightIn, 0.4)` |
| **Parameter: New column name** | `'approxMedianIn'` |

**Percentile:** This transformation calculates the 68th percentile.

| | |
|---|---|
| **Transformation Name** | `New formula` |
| **Parameter: Formula type** | `Single row formula` |
| **Parameter: Formula** | `percentile(heightIn, 68, linear)` |
| **Parameter: New column name** | `'percentile68In'` |

| | |
|---|---|
| **Transformation Name** | `New formula` |
| **Parameter: Formula type** | `Single row formula` |
| **Parameter: Formula** | `approximatepercentile(heightIn, 68, 0.4)` |
| **Parameter: New column name** | `'approxPercentile68In'` |

**Quartile:** This transformation calculates the first quartile, which corresponds to the 25th percentile.

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | quartile(heightIn, 1, linear) |
| Parameter: New column name | 'percentile25In' |

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | approximatequartile(heightIn, 1, 0.4) |
| Parameter: New column name | 'approxPercentile25In' |

**Results:**

| studentId | heightIn | approxPercentile25In | percentile25In | approxPercentile68In | percentile68In | approxMedianIn | |
|---|---|---|---|---|---|---|---|
| 1 | 64 | 64 | 64 | 67.1 | 66.92 | 66 | 6 |
| 2 | 65 | 64 | 64 | 67.1 | 66.92 | 66 | 6 |
| 3 | 63 | 64 | 64 | 67.1 | 66.92 | 66 | 6 |
| 4 | 64 | 64 | 64 | 67.1 | 66.92 | 66 | 6 |
| 5 | 62 | 64 | 64 | 67.1 | 66.92 | 66 | 6 |
| 6 | 66 | 64 | 64 | 67.1 | 66.92 | 66 | 6 |
| 7 | 66 | 64 | 64 | 67.1 | 66.92 | 66 | 6 |
| 8 | 65 | 64 | 64 | 67.1 | 66.92 | 66 | 6 |
| 9 | 69 | 64 | 64 | 67.1 | 66.92 | 66 | 6 |
| 10 | 66 | 64 | 64 | 67.1 | 66.92 | 66 | 6 |
| 11 | 73 | 64 | 64 | 67.1 | 66.92 | 66 | 6 |
| 12 | 69 | 64 | 64 | 67.1 | 66.92 | 66 | 6 |
| 13 | 69 | 64 | 64 | 67.1 | 66.92 | 66 | 6 |
| 14 | 61 | 64 | 64 | 67.1 | 66.92 | 66 | 6 |
| 15 | 64 | 64 | 64 | 67.1 | 66.92 | 66 | 6 |
| 16 | 61 | 64 | 64 | 67.1 | 66.92 | 66 | 6 |
| 17 | 71 | 64 | 64 | 67.1 | 66.92 | 66 | 6 |
| 18 | 67 | 64 | 64 | 67.1 | 66.92 | 66 | 6 |
| 19 | 73 | 64 | 64 | 67.1 | 66.92 | 66 | 6 |
| 20 | 66 | 64 | 64 | 67.1 | 66.92 | 66 | 6 |

# STDEV Function

Computes the standard deviation across all column values of Integer or Decimal type.

The **standard deviation** of a set of values attempts to measure the spread in values around the mean and is used to measure confidence in statistical results. A standard deviation of zero means that all values are the same, and a small standard deviation means that the values are closely bunched together. A high value for standard deviation indicates that the numbers are spread out widely. Standard deviation is always a positive value.

- If a row contains a missing or null value, it is not factored into the calculation.
- If no numeric values are detected in the input column, the function returns `0`.

**Relevant terms:**

| Term | Description |
|---|---|
| Population | Population statistical functions are computed from all possible values. See *https://en.wikipedia.org/wiki/Statistical_population*. |
| Sample | Sample-based statistical functions are computed from a subset or sample of all values. See *https://en.wikipedia.org/wiki/Sampling_(statistics)*.<br><br>These function names include $\mathrm{SAMP}$ in their name.<br><br>> **NOTE:** Statistical sampling has no relationship to the samples taken within the product. When statistical functions are computed during job execution, they are applied across the entire dataset. Sample method calculations are computed at that time. |

- This function is calculated across the entire population.
- For more information on a sampled version of this function, see *STDEVSAMP Function*.

The square of standard deviation is variance. See *VAR Function*.

For a version of this function computed over a rolling window of rows, see *ROLLINGSTDEV Function*.

**Wrangle vs. SQL:** This function is part of Wrangle , a proprietary data transformation language. Wrangle is not SQL. For more information, see *Wrangle Language*.

## Basic Usage

```
stdev(myRating)
```

**Output:** Returns the standard deviation of the values from the `myRating` column.

## Syntax and Arguments

```
stdev(function_col_ref) [group:group_col_ref] [limit:limit_count]
```

| Argument | Required? | Data Type | Description |
|---|---|---|---|
| function_col_ref | Y | string | Name of column to which to apply the function |

For more information on the `group` and `limit` parameters, see *Pivot Transform*.

For more information on syntax standards, see *Language Documentation Syntax Notes*.

**function_col_ref**

Name of the column the values of which you want to calculate the variance. Column must contain Integer or Decimal values.

- Literal values are not supported as inputs.
- Multiple columns and wildcards are not supported.

**Usage Notes:**

| Required? | Data Type | Example Value |
|---|---|---|
| Yes | String (column reference) | `myValues` |

## Examples

> **Tip:** For additional examples, see *Common Tasks*.

This example illustrates how you can apply statistical functions to your dataset. Calculations include average (mean), max, min, standard deviation, and variance.

**Functions:**

| Item | Description |
|---|---|
| AVERAGE Function | Computes the average (mean) from all row values in a column or group. Input column can be of Integer or Decimal. |
| MIN Function | Computes the minimum value found in all row values in a column. Input column can be of Integer, Decimal or Datetime. |
| MAX Function | Computes the maximum value found in all row values in a column. Inputs can be Integer, Decimal, or Datetime. |
| VAR Function | Computes the variance among all values in a column. Input column can be of Integer or Decimal. If no numeric values are detected in the input column, the function returns $0$. |
| STDEV Function | Computes the standard deviation across all column values of Integer or Decimal type. |
| NUMFORMAT Function | Formats a numeric set of values according to the specified number formatting. Source values can be a literal numeric value, a function returning a numeric value, or reference to a column containing an Integer or Decimal values. |
| MODE Function | Computes the mode (most frequent value) from all row values in a column, according to their grouping. Input column can be of Integer, Decimal, or Datetime type. |

**Source:**

Students took a test and recorded the following scores. You want to perform some statistical analysis on them:

| Student | Score |
|---|---|
| Anna | 84 |
| Ben | 71 |

| | |
|---|---|
| Caleb | 76 |
| Danielle | 87 |
| Evan | 85 |
| Faith | 92 |
| Gabe | 85 |
| Hannah | 99 |
| Ian | 73 |
| Jane | 68 |

**Transformation:**

You can use the following transformations to calculate the average (mean), minimum, and maximum scores:

| | |
|---|---|
| **Transformation Name** | `New formula` |
| **Parameter: Formula type** | `Single row formula` |
| **Parameter: Formula** | `AVERAGE(Score)` |
| **Parameter: New column name** | `'avgScore'` |

| | |
|---|---|
| **Transformation Name** | `New formula` |
| **Parameter: Formula type** | `Single row formula` |
| **Parameter: Formula** | `MIN(Score)` |
| **Parameter: New column name** | `'minScore'` |

| | |
|---|---|
| **Transformation Name** | `New formula` |
| **Parameter: Formula type** | `Single row formula` |
| **Parameter: Formula** | `MAX(Score)` |
| **Parameter: New column name** | `'maxScore'` |

To apply statistical functions to your data, you can use the `VAR` and `STDEV` functions, which can be used as the basis for other statistical calculations.

| | |
|---|---|
| **Transformation Name** | `New formula` |
| **Parameter: Formula type** | `Single row formula` |
| **Parameter: Formula** | `VAR(Score)` |
| **Parameter: New column name** | `var_Score` |

| | |
|---|---|
| **Transformation Name** | `New formula` |
| **Parameter: Formula type** | `Single row formula` |

| Parameter: Formula | `STDEV(Score)` |
|---|---|
| Parameter: New column name | `stdev_Score` |

For each score, you can now calculate the variation of each one from the average, using the following:

| Transformation Name | `New formula` |
|---|---|
| Parameter: Formula type | `Single row formula` |
| Parameter: Formula | `((Score - avg_Score) / stdev_Score)` |
| Parameter: New column name | `'stDevs'` |

Now, you want to apply grades based on a formula:

| Grade | standard deviations from avg (stDevs) |
|---|---|
| A | stDevs > 1 |
| B | stDevs > 0.5 |
| C | -1 <= stDevs <= 0.5 |
| D | stDevs < -1 |
| F | stDevs < -2 |

You can build the following transformation using the `IF` function to calculate grades.

| Transformation Name | `New formula` |
|---|---|
| Parameter: Formula type | `Single row formula` |
| Parameter: Formula | `IF((stDevs > 1),'A',IF((stDevs < -2),'F',IF((stDevs < -1),'D',IF` `((stDevs > 0.5),'B','C'))))` |

To clean up the content, you might want to apply some formatting to the score columns. The following reformats the `stdev_Score` and `stDevs` columns to display two decimal places:

| Transformation Name | `Edit column with formula` |
|---|---|
| Parameter: Columns | `stdev_Score` |
| Parameter: Formula | `NUMFORMAT(stdev_Score, '##.00')` |

| Transformation Name | `Edit column with formula` |
|---|---|
| Parameter: Columns | `stDevs` |
| Parameter: Formula | `NUMFORMAT(stDevs, '##.00')` |

| Transformation Name | `New formula` |
|---|---|
| Parameter: Formula type | `Single row formula` |

| Parameter: Formula | `MODE(Score)` |
|---|---|
| Parameter: New column name | `'modeScore'` |

**Results:**

| Student | Score | modeScore | avgScore | minScore | maxScore | var_Score | stdev_Score | stDevs | Grade |
|---|---|---|---|---|---|---|---|---|---|
| Anna | 84 | 85 | 82 | 68 | 99 | 87.000000000 00001 | 9.33 | 0.21 | C |
| Ben | 71 | 85 | 82 | 68 | 99 | 87.000000000 00001 | 9.33 | -1.18 | D |
| Caleb | 76 | 85 | 82 | 68 | 99 | 87.000000000 00001 | 9.33 | -0.64 | C |
| Danielle | 87 | 85 | 82 | 68 | 99 | 87.000000000 00001 | 9.33 | 0.54 | B |
| Evan | 85 | 85 | 82 | 68 | 99 | 87.000000000 00001 | 9.33 | 0.32 | C |
| Faith | 92 | 85 | 82 | 68 | 99 | 87.000000000 00001 | 9.33 | 1.07 | A |
| Gabe | 85 | 85 | 82 | 68 | 99 | 87.000000000 00001 | 9.33 | 0.32 | C |
| Hannah | 99 | 85 | 82 | 68 | 99 | 87.000000000 00001 | 9.33 | 1.82 | A |
| Ian | 73 | 85 | 82 | 68 | 99 | 87.000000000 00001 | 9.33 | -0.96 | C |
| Jane | 68 | 85 | 82 | 68 | 99 | 87.000000000 00001 | 9.33 | -1.50 | D |

# STDEVIF Function

**Contents:**

Generates the standard deviation of values by group in a column that meet a specific condition.

> **NOTE:** When added to a transform, this function is applied to the sample in the data grid. If you change your sample or run the job, the computed values for this function are updated. Transforms that change the number of rows in subsequent recipe steps do not affect the values computed for this step.

**Relevant terms:**

| Term | Description |
|------|-------------|
| Population | Population statistical functions are computed from all possible values. See *https://en.wikipedia.org/wiki/Statistical_population*. |
| Sample | Sample-based statistical functions are computed from a subset or sample of all values. See *https://en.wikipedia.org/wiki/Sampling_(statistics)*. <br><br>These function names include SAMP in their name. <br><br> > **NOTE:** Statistical sampling has no relationship to the samples taken within the product. When statistical functions are computed during job execution, they are applied across the entire dataset. Sample method calculations are computed at that time. |

- This function is calculated across the entire population.
- For more information on a sampled version of this function, see *STDEVSAMPIF Function*.

**Wrangle vs. SQL:** This function is part of Wrangle , a proprietary data transformation language. Wrangle is not SQL. For more information, see *Wrangle Language*.

## Basic Usage

```
stdevif(testScores, testScores &gt; 0)
```

**Output:** Returns the standard deviation of the `testScores` column when the `testScores` value is greater than 0.

## Syntax and Arguments

```
stdevif(col_ref, test_expression) [group:group_col_ref] [limit:limit_count]
```

| Argument | Required? | Data Type | Description |
|---|---|---|---|
| col_ref | Y | string | Reference to the column you wish to evaluate. |
| test_expression | Y | string | Expression that is evaluated. Must resolve to `true` or `false` |

For more information on syntax standards, see *Language Documentation Syntax Notes*.

For more information on the `group` and `limit` parameters, see *Pivot Transform*.

### col_ref

Name of the column whose values you wish to use in the calculation. Column must be a numeric (Integer or Decimal) type.

**Usage Notes:**

| Required? | Data Type | Example Value |
|---|---|---|
| Yes | String that corresponds to the name of the column | `myValues` |

### test_expression

This parameter contains the expression to evaluate. This expression must resolve to a Boolean (`true` or `false`) value.

**Usage Notes:**

| Required? | Data Type | Example Value |
|---|---|---|
| Yes | String expression that evaluates to `true` or `false` | `(LastName == 'Mouse' && FirstName == 'Mickey')` |

## Examples

> **Tip:** For additional examples, see *Common Tasks*.

### Example - Conditional Calculation Functions

This example illustrates how to use the conditional calculation functions.

**Functions**:

| Item | Description |
|---|---|
| AVERAGEIF Function | Generates the average value of rows in each group that meet a specific condition. Generated value is of Decimal type. |
| MINIF Function | Generates the minimum value of rows in each group that meet a specific condition. Inputs can be Integer, Decimal, or Datetime. |
| MAXIF Function | Generates the maximum value of rows in each group that meet a specific condition. Inputs can be Integer, Decimal, or Datetime. |
| VARIF Function | Generates the variance of values by group in a column that meet a specific condition. |

| STDEVIF Function | Generates the standard deviation of values by group in a column that meet a specific condition. |
|---|---|

**Source:**

Here is some example weather data:

| date | city | rain | temp | wind |
|---|---|---|---|---|
| 1/23/17 | Valleyville | 0.00 | 12.8 | 6.7 |
| 1/23/17 | Center Town | 0.31 | 9.4 | 5.3 |
| 1/23/17 | Magic Mountain | 0.00 | 0.0 | 7.3 |
| 1/24/17 | Valleyville | 0.25 | 17.2 | 3.3 |
| 1/24/17 | Center Town | 0.54 | 1.1 | 7.6 |
| 1/24/17 | Magic Mountain | 0.32 | 5.0 | 8.8 |
| 1/25/17 | Valleyville | 0.02 | 3.3 | 6.8 |
| 1/25/17 | Center Town | 0.83 | 3.3 | 5.1 |
| 1/25/17 | Magic Mountain | 0.59 | -1.7 | 6.4 |
| 1/26/17 | Valleyville | 1.08 | 15.0 | 4.2 |
| 1/26/17 | Center Town | 0.96 | 6.1 | 7.6 |
| 1/26/17 | Magic Mountain | 0.77 | -3.9 | 3.0 |
| 1/27/17 | Valleyville | 1.00 | 7.2 | 2.8 |
| 1/27/17 | Center Town | 1.32 | 20.0 | 0.2 |
| 1/27/17 | Magic Mountain | 0.77 | 5.6 | 5.2 |
| 1/28/17 | Valleyville | 0.12 | -6.1 | 5.1 |
| 1/28/17 | Center Town | 0.14 | 5.0 | 4.9 |
| 1/28/17 | Magic Mountain | 1.50 | 1.1 | 0.4 |
| 1/29/17 | Valleyville | 0.36 | 13.3 | 7.3 |
| 1/29/17 | Center Town | 0.75 | 6.1 | 9.0 |
| 1/29/17 | Magic Mountain | 0.60 | 3.3 | 6.0 |

**Transformation:**

The following computes average temperature for rainy days by city:

| Transformation Name | New formula |
|---|---|
| **Parameter: Formula type** | Single row formula |
| **Parameter: Formula** | AVERAGEIF(temp, rain > 0) |
| **Parameter: Group rows by** | city |
| **Parameter: New column name** | 'avgTempWRain' |

The following computes maximum wind for sub-zero days by city:

| Transformation Name | New formula |
|---|---|

| Parameter: Formula type | Single row formula |
|---|---|
| Parameter: Formula | MAXIF(wind,temp < 0) |
| Parameter: Group rows by | city |
| Parameter: New column name | 'maxWindSubZero' |

This step calculates the minimum temp when the wind is less than 5 mph by city:

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | MINIF(temp,wind<5) |
| Parameter: Group rows by | city |
| Parameter: New column name | 'minTempWind5' |

This step computes the variance in temperature for rainy days by city:

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | VARIF(temp,rain >0) |
| Parameter: Group rows by | city |
| Parameter: New column name | 'varTempWRain' |

The following computes the standard deviation in rainfall for Center Town:

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | STDEVIF(rain,city=='Center Town') |
| Parameter: Group rows by | city |
| Parameter: New column name | 'stDevRainCT' |

You can use the following transforms to format the generated output. Note the $col placeholder value for the multi-column transforms:

| Transformation Name | Edit column with formula |
|---|---|
| Parameter: Columns | stDevRainCenterTown,maxWindSubZero |
| Parameter: Formula | numformat($col,'##.##') |

Since the following rely on data that has only one significant digit, you should format them differently:

| Transformation Name | Edit column with formula |
|---|---|
| Parameter: Columns | varTempWRain,avgTempWRain,minTempWind5 |
| Parameter: Formula | numformat($col,'##.#') |

**Results:**

| date | city | rain | temp | wind | avgTempWRain | maxWindSubZero | minTempWind5 | varTempWRain | stDevRain |
|---|---|---|---|---|---|---|---|---|---|
| 1/23/17 | Valley ville | 0.00 | 12.8 | 6.7 | 8.3 | 5.1 | 7.2 | 63.8 | 0.37 |
| 1/23/17 | Center Town | 0.31 | 9.4 | 5.3 | 7.3 | | 5 | 32.6 | 0.37 |
| 1/23/17 | Magic Mountain | 0.00 | 0.0 | 7.3 | 1.6 | 6.43 | -3.9 | 12 | 0.37 |
| 1/24/17 | Valley ville | 0.25 | 17.2 | 3.3 | 8.3 | 5.1 | 7.2 | 63.8 | 0.37 |
| 1/24/17 | Center Town | 0.54 | 1.1 | 7.6 | 7.3 | | 5 | 32.6 | 0.37 |
| 1/24/17 | Magic Mountain | 0.32 | 5.0 | 8.8 | 1.6 | 6.43 | -3.9 | 12 | 0.37 |
| 1/25/17 | Valley ville | 0.02 | 3.3 | 6.8 | 8.3 | 5.1 | 7.2 | 63.8 | 0.37 |
| 1/25/17 | Center Town | 0.83 | 3.3 | 5.1 | 7.3 | | 5 | 32.6 | 0.37 |
| 1/25/17 | Magic Mountain | 0.59 | -1.7 | 6.4 | 1.6 | 6.43 | -3.9 | 12 | 0.37 |
| 1/26/17 | Valley ville | 1.08 | 15.0 | 4.2 | 8.3 | 5.1 | 7.2 | 63.8 | 0.37 |
| 1/26/17 | Center Town | 0.96 | 6.1 | 7.6 | 7.3 | | 5 | 32.6 | 0.37 |
| 1/26/17 | Magic Mountain | 0.77 | -3.9 | 3.0 | 1.6 | 6.43 | -3.9 | 12 | 0.37 |
| 1/27/17 | Valley ville | 1.00 | 7.2 | 2.8 | 8.3 | 5.1 | 7.2 | 63.8 | 0.37 |
| 1/27/17 | Center Town | 1.32 | 20.0 | 0.2 | 7.3 | | 5 | 32.6 | 0.37 |
| 1/27/17 | Magic Mountain | 0.77 | 5.6 | 5.2 | 1.6 | 6.43 | -3.9 | 12 | 0.37 |
| 1/28/17 | Valley ville | 0.12 | -6.1 | 5.1 | 8.3 | 5.1 | 7.2 | 63.8 | 0.37 |
| 1/28/17 | Center Town | 0.14 | 5.0 | 4.9 | 7.3 | | 5 | 32.6 | 0.37 |
| 1/28/17 | Magic Mountain | 1.50 | 1.1 | 0.4 | 1.6 | 6.43 | -3.9 | 12 | 0.37 |

| 1/29/17 | Valley ville | 0.36 | 13.3 | 7.3 | 8.3 | 5.1 | | 7.2 | 63.8 | 0.37 |
| 1/29/17 | Center Town | 0.75 | 6.1 | 9.0 | 7.3 | | | 5 | 32.6 | 0.37 |
| 1/29/17 | Magic Mountain | 0.60 | 3.3 | 6.0 | 1.6 | 6.43 | | -3.9 | 12 | 0.37 |

# STDEVSAMP Function

Computes the standard deviation across column values of Integer or Decimal type using the sample statistical method.

The **standard deviation** of a set of values attempts to measure the spread in values around the mean and is used to measure confidence in statistical results. A standard deviation of zero means that all values are the same, and a small standard deviation means that the values are closely bunched together. A high value for standard deviation indicates that the numbers are spread out widely. Standard deviation is always a positive value.

> **NOTE:** This function applies to a sample of the entire population. More information is below.

**Relevant terms:**

| Term | Description |
|------|-------------|
| Population | Population statistical functions are computed from all possible values. See *https://en.wikipedia.org/wiki/Statistical_population*. |
| Sample | Sample-based statistical functions are computed from a subset or sample of all values. See *https://en.wikipedia.org/wiki/Sampling_(statistics)*.<br><br>These function names include $SAMP$ in their name.<br><br>> **NOTE:** Statistical sampling has no relationship to the samples taken within the product. When statistical functions are computed during job execution, they are applied across the entire dataset. Sample method calculations are computed at that time. |

- This function is calculated across a sample of all values.
- For more information on a population version of this function, see *STDEV Function*.

If a row contains a missing or null value, it is not factored into the calculation. If no numeric values are detected in the input column, the function returns `0`.

The square of standard deviation is variance. See *VAR Function*.

For a version of this function computed over a rolling window of rows, see *ROLLINGSTDEV Function*.

**Wrangle vs. SQL:** This function is part of Wrangle , a proprietary data transformation language. Wrangle is not SQL. For more information, see *Wrangle Language*.

## Basic Usage

```
stdevsamp(myRating)
```

**Output:** Returns the standard deviation of the values from the `myRating` column using the sample method of calculation.

## Syntax and Arguments

```
stdevsamp(col_ref) [group:group_col_ref] [limit:limit_count]
```

| Argument | Required? | Data Type | Description |
|---|---|---|---|
| function_col_ref | Y | string | Name of column to which to apply the function |

For more information on the `group` and `limit` parameters, see *Pivot Transform*.

For more information on syntax standards, see *Language Documentation Syntax Notes*.

**col_ref**

Name of the column whose values you wish to use in the calculation. Column must be a numeric (Integer or Decimal) type.

- Literal values are not supported as inputs.
- Multiple columns and wildcards are not supported.

**Usage Notes:**

| Required? | Data Type | Example Value |
|---|---|---|
| Yes | String (column reference) | `myValues` |

## Examples

> **Tip:** For additional examples, see *Common Tasks*.

This example shows some of the statistical functions that use the sample method of computation.

**Functions**:

| Item | Description |
|---|---|
| STDEVSAMP Function | Computes the standard deviation across column values of Integer or Decimal type using the sample statistical method. |
| VARSAMP Function | Computes the variance among all values in a column using the sample statistical method. Input column can be of Integer or Decimal. If no numeric values are detected in the input column, the function returns `0`. |
| STDEVSAMP IF Function | Generates the standard deviation of values by group in a column that meet a specific condition using the sample statistical method. |
| VARSAMPIF Function | Generates the variance of values by group in a column that meet a specific condition using the sample statistical method. |
| ROUND Function | Rounds input value to the nearest integer. Input can be an Integer, a Decimal, a column reference, or an expression. Optional second argument can be used to specify the number of digits to which to round. |

**Source:**

Students took tests on three consecutive Saturdays:

| Student | Date | Score |
|---|---|---|
| Andrew | 11/9/19 | 81 |

| | | |
|---|---|---|
| Bella | 11/9/19 | 84 |
| Christina | 11/9/19 | 79 |
| David | 11/9/19 | 64 |
| Ellen | 11/9/19 | 61 |
| Fred | 11/9/19 | 63 |
| Andrew | 11/16/19 | 73 |
| Bella | 11/16/19 | 88 |
| Christina | 11/16/19 | 78 |
| David | 11/16/19 | 67 |
| Ellen | 11/16/19 | 87 |
| Fred | 11/16/19 | 90 |
| Andrew | 11/23/19 | 76 |
| Bella | 11/23/19 | 93 |
| Christina | 11/23/19 | 81 |
| David | 11/23/19 | 97 |
| Ellen | 11/23/19 | 97 |
| Fred | 11/23/19 | 91 |

**Transformation:**

You can use the following transformations to calculate standard deviation and variance across all dates using the sample method. Each computation has been rounded to three digits.

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | round(stdevsamp(Score), 3) |
| Parameter: New column name | 'stdevSamp' |

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | round(varsamp(Score), 3) |
| Parameter: New column name | 'varSamp' |

You can use the following to limit the previous statistical computations to the last two Saturdays of testing:

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | round(stdevsampif(Score, Date != '11\/9\/2019'), 3) |
| Parameter: New column name | 'stdevSampIf' |

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | round(varsampif(Score, Date != '11\/9\/2019'), 3) |
| Parameter: New column name | 'varSampIf' |

**Results:**

| Student | Date | Score | varSampIf | stdevSampIf | varSamp | stdevSamp |
|---|---|---|---|---|---|---|
| Andrew | 11/9/19 | 81 | 94.515 | 9.722 | 131.673 | 11.475 |
| Bella | 11/9/19 | 84 | 94.515 | 9.722 | 131.673 | 11.475 |
| Christina | 11/9/19 | 79 | 94.515 | 9.722 | 131.673 | 11.475 |
| David | 11/9/19 | 64 | 94.515 | 9.722 | 131.673 | 11.475 |
| Ellen | 11/9/19 | 61 | 94.515 | 9.722 | 131.673 | 11.475 |
| Fred | 11/9/19 | 63 | 94.515 | 9.722 | 131.673 | 11.475 |
| Andrew | 11/16/19 | 73 | 94.515 | 9.722 | 131.673 | 11.475 |
| Bella | 11/16/19 | 88 | 94.515 | 9.722 | 131.673 | 11.475 |
| Christina | 11/16/19 | 78 | 94.515 | 9.722 | 131.673 | 11.475 |
| David | 11/16/19 | 67 | 94.515 | 9.722 | 131.673 | 11.475 |
| Ellen | 11/16/19 | 87 | 94.515 | 9.722 | 131.673 | 11.475 |
| Fred | 11/16/19 | 90 | 94.515 | 9.722 | 131.673 | 11.475 |
| Andrew | 11/23/19 | 76 | 94.515 | 9.722 | 131.673 | 11.475 |
| Bella | 11/23/19 | 93 | 94.515 | 9.722 | 131.673 | 11.475 |
| Christina | 11/23/19 | 81 | 94.515 | 9.722 | 131.673 | 11.475 |
| David | 11/23/19 | 97 | 94.515 | 9.722 | 131.673 | 11.475 |
| Ellen | 11/23/19 | 97 | 94.515 | 9.722 | 131.673 | 11.475 |
| Fred | 11/23/19 | 91 | 94.515 | 9.722 | 131.673 | 11.475 |

# STDEVSAMPIF Function

**Contents:**

Generates the standard deviation of values by group in a column that meet a specific condition using the sample statistical method.

> **NOTE:** When added to a transform, this function is applied to the sample in the data grid. If you change your sample or run the job, the computed values for this function are updated. Transforms that change the number of rows in subsequent recipe steps do not affect the values computed for this step.

> **NOTE:** This function applies to a sample of the entire population. More information is below.

**Relevant terms:**

| Term | Description |
|------|-------------|
| Population | Population statistical functions are computed from all possible values. See *https://en.wikipedia.org/wiki/Statistical_population*. |
| Sample | Sample-based statistical functions are computed from a subset or sample of all values. See *https://en.wikipedia.org/wiki/Sampling_(statistics)*. <br><br> These function names include $SAMP$ in their name. <br><br> > **NOTE:** Statistical sampling has no relationship to the samples taken within the product. When statistical functions are computed during job execution, they are applied across the entire dataset. Sample method calculations are computed at that time. |

- This function is calculated across a sample of all values.
- For more information on a population version of this function, see *STDEVIF Function*.

**Wrangle vs. SQL:** This function is part of Wrangle , a proprietary data transformation language. Wrangle is not SQL. For more information, see *Wrangle Language*.

## Basic Usage

```
<span>stdevsampif</span>(testScores, testScores &gt; 0)
```

**Output:** Returns the standard deviation of the `testScores` column when the `testScores` value is greater than 0.

## Syntax and Arguments

```
stdevsampif(col_ref, test_expression) [group:group_col_ref] [limit:limit_count]
```

| Argument | Required? | Data Type | Description |
|---|---|---|---|
| col_ref | Y | string | Reference to the column you wish to evaluate. |
| test_expression | Y | string | Expression that is evaluated. Must resolve to `true` or `false` |

For more information on syntax standards, see *Language Documentation Syntax Notes*.

For more information on the `group` and `limit` parameters, see *Pivot Transform*.

### col_ref

Name of the column whose values you wish to use in the calculation. Column must be a numeric (Integer or Decimal) type.

**Usage Notes:**

| Required? | Data Type | Example Value |
|---|---|---|
| Yes | String that corresponds to the name of the column | `myValues` |

### test_expression

This parameter contains the expression to evaluate. This expression must resolve to a Boolean (`true` or `false`) value.

**Usage Notes:**

| Required? | Data Type | Example Value |
|---|---|---|
| Yes | String expression that evaluates to `true` or `false` | `(LastName == 'Mouse' && FirstName == 'Mickey')` |

## Examples

> **Tip:** For additional examples, see *Common Tasks*.

**Example - Conditional Calculation Functions**

This example shows some of the statistical functions that use the sample method of computation.

**Functions**:

| Item | Description |
|---|---|
| STDEVSAMP Function | Computes the standard deviation across column values of Integer or Decimal type using the sample statistical method. |

| | |
|---|---|
| VARSAMP Function | Computes the variance among all values in a column using the sample statistical method. Input column can be of Integer or Decimal. If no numeric values are detected in the input column, the function returns 0. |
| STDEVSAMP IF Function | Generates the standard deviation of values by group in a column that meet a specific condition using the sample statistical method. |
| VARSAMPIF Function | Generates the variance of values by group in a column that meet a specific condition using the sample statistical method. |
| ROUND Function | Rounds input value to the nearest integer. Input can be an Integer, a Decimal, a column reference, or an expression. Optional second argument can be used to specify the number of digits to which to round. |

**Source:**

Students took tests on three consecutive Saturdays:

| Student | Date | Score |
|---|---|---|
| Andrew | 11/9/19 | 81 |
| Bella | 11/9/19 | 84 |
| Christina | 11/9/19 | 79 |
| David | 11/9/19 | 64 |
| Ellen | 11/9/19 | 61 |
| Fred | 11/9/19 | 63 |
| Andrew | 11/16/19 | 73 |
| Bella | 11/16/19 | 88 |
| Christina | 11/16/19 | 78 |
| David | 11/16/19 | 67 |
| Ellen | 11/16/19 | 87 |
| Fred | 11/16/19 | 90 |
| Andrew | 11/23/19 | 76 |
| Bella | 11/23/19 | 93 |
| Christina | 11/23/19 | 81 |
| David | 11/23/19 | 97 |
| Ellen | 11/23/19 | 97 |
| Fred | 11/23/19 | 91 |

**Transformation:**

You can use the following transformations to calculate standard deviation and variance across all dates using the sample method. Each computation has been rounded to three digits.

| | |
|---|---|
| **Transformation Name** | New formula |
| **Parameter: Formula type** | Single row formula |
| **Parameter: Formula** | round(stdevsamp(Score), 3) |

| Parameter: New column name | 'stdevSamp' |
|---|---|

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | round(varsamp(Score), 3) |
| Parameter: New column name | 'varSamp' |

You can use the following to limit the previous statistical computations to the last two Saturdays of testing:

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | round(stdevsampif(Score, Date != '11\/9\/2019'), 3) |
| Parameter: New column name | 'stdevSampIf' |

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | round(varsampif(Score, Date != '11\/9\/2019'), 3) |
| Parameter: New column name | 'varSampIf' |

**Results:**

| Student | Date | Score | varSampIf | stdevSampIf | varSamp | stdevSamp |
|---|---|---|---|---|---|---|
| Andrew | 11/9/19 | 81 | 94.515 | 9.722 | 131.673 | 11.475 |
| Bella | 11/9/19 | 84 | 94.515 | 9.722 | 131.673 | 11.475 |
| Christina | 11/9/19 | 79 | 94.515 | 9.722 | 131.673 | 11.475 |
| David | 11/9/19 | 64 | 94.515 | 9.722 | 131.673 | 11.475 |
| Ellen | 11/9/19 | 61 | 94.515 | 9.722 | 131.673 | 11.475 |
| Fred | 11/9/19 | 63 | 94.515 | 9.722 | 131.673 | 11.475 |
| Andrew | 11/16/19 | 73 | 94.515 | 9.722 | 131.673 | 11.475 |
| Bella | 11/16/19 | 88 | 94.515 | 9.722 | 131.673 | 11.475 |
| Christina | 11/16/19 | 78 | 94.515 | 9.722 | 131.673 | 11.475 |
| David | 11/16/19 | 67 | 94.515 | 9.722 | 131.673 | 11.475 |
| Ellen | 11/16/19 | 87 | 94.515 | 9.722 | 131.673 | 11.475 |
| Fred | 11/16/19 | 90 | 94.515 | 9.722 | 131.673 | 11.475 |
| Andrew | 11/23/19 | 76 | 94.515 | 9.722 | 131.673 | 11.475 |
| Bella | 11/23/19 | 93 | 94.515 | 9.722 | 131.673 | 11.475 |
| Christina | 11/23/19 | 81 | 94.515 | 9.722 | 131.673 | 11.475 |

| David | 11/23/19 | 97 | 94.515 | 9.722 | 131.673 | 11.475 |
| Ellen | 11/23/19 | 97 | 94.515 | 9.722 | 131.673 | 11.475 |
| Fred | 11/23/19 | 91 | 94.515 | 9.722 | 131.673 | 11.475 |

# SUM Function

Computes the sum of all values found in all row values in a column. Input column can be of Integer or Decimal.

- If a row contains a missing or null value, it is not factored into the calculation. If no numeric values are found in the source column, the function returns `0` .
- When used in a `pivot` transform, the function is computed for each instance of the value specified in the `group` parameter. See *Pivot Transform*.

For a version of this function computed over a rolling window of rows, see *ROLLINGSUM Function*.

**Wrangle vs. SQL:** This function is part of Wrangle , a proprietary data transformation language. Wrangle is not SQL. For more information, see *Wrangle Language*.

## Basic Usage

```
sum(myRating)
```

**Output:** Returns the sum of the group of values from the `myRating` column.

**Wrangle vs. SQL:** This function is part of Wrangle , a proprietary data transformation language. Wrangle is not SQL. For more information, see *Wrangle Language*.

## Syntax and Arguments

```
sum(function_col_ref) [group:group_col_ref] [limit:limit_count]
```

| Argument | Required? | Data Type | Description |
|---|---|---|---|
| function_col_ref | Y | string | Name of column to which to apply the function |

For more information on the `group` and `limit` parameters, see *Pivot Transform*.

For more information on syntax standards, see *Language Documentation Syntax Notes*.

**function_col_ref**

Name of the column the values of which you want to calculate the sum. Column must contain Integer or Decimal values.

- Literal values are not supported as inputs.
- Multiple columns and wildcards are not supported.

**Usage Notes:**

| Required? | Data Type | Example Value |
|---|---|---|
| Yes | String (column reference) | `myValues` |

# Examples

> **Tip:** For additional examples, see *Common Tasks*.

This example demonstrates you to extract values from one column of an array into a new column.

**Functions**:

| Item | Description |
|------|-------------|
| LIST Function | Extracts the set of values from a column into an array stored in a new column. This function is typically part of an aggregation. |
| UNIQUE Function | Extracts the set of unique values from a column into an array stored in a new column. This function is typically part of an aggregation. |
| DATEFORMAT Function | Formats a specified Datetime set of values according to the specified date format. Source values can be a reference to a column containing Datetime values. |

You have the following set of orders for two months, and you are interested in identifying the set of colors that have been sold for each product for each month and the total quantity of product sold for each month.

**Source:**

| OrderId | Date | Item | Qty | Color |
|---------|---------|-------|-----|--------|
| 1001 | 1/15/15 | Pants | 1 | red |
| 1002 | 1/15/15 | Shirt | 2 | green |
| 1003 | 1/15/15 | Hat | 3 | blue |
| 1004 | 1/16/15 | Shirt | 4 | yellow |
| 1005 | 1/16/15 | Hat | 5 | red |
| 1006 | 1/20/15 | Pants | 6 | green |
| 1007 | 1/15/15 | Hat | 7 | blue |
| 1008 | 4/15/15 | Shirt | 8 | yellow |
| 1009 | 4/15/15 | Shoes | 9 | brown |
| 1010 | 4/16/15 | Pants | 1 | red |
| 1011 | 4/16/15 | Hat | 2 | green |
| 1012 | 4/16/15 | Shirt | 3 | blue |
| 1013 | 4/20/15 | Shoes | 4 | black |
| 1014 | 4/20/15 | Hat | 5 | blue |
| 1015 | 4/20/15 | Pants | 6 | black |

**Transformation:**

To track by month, you need a column containing the month value extracted from the date:

| Transformation Name | Edit column with formula |
|---------------------|--------------------------|

| Parameter: Columns | Date |
|---|---|
| Parameter: Formula | DATEFORMAT(Date, 'MMM yyyy') |

You can use the following transform to check the list of unique values among the colors:

| Transformation Name | Pivot columns |
|---|---|
| Parameter: Row labels | Date |
| Parameter: Values | unique(Color, 1000) |
| Parameter: Max number of columns to create | 10 |

| Date | unique_Color |
|---|---|
| Jan 2015 | ["green","blue","red","yellow"] |
| Apr 2015 | ["brown","blue","red","yellow","black","green"] |

Delete the above transform.

You can aggregate the data in your dataset, grouped by the reformatted `Date` values, and apply the `LIST` function to the `Color` column. In the same aggregation, you can include a summation function for the `Qty` column:

| Transformation Name | Pivot columns |
|---|---|
| Parameter: Row labels | Date |
| Parameter: Values | list(Color, 1000),sum(Qty) |
| Parameter: Max number of columns to create | 10 |

**Results:**

| Date | list_Color | sum_Qty |
|---|---|---|
| Jan 2015 | ["green","blue","blue","red","green","red","yellow"] | 28 |
| Apr 2015 | ["brown","blue","red","yellow","black","blue","black","green"] | 38 |

# SUMIF Function

**Contents:**

---

Generates the sum of rows in each group that meet a specific condition.

> **NOTE:** When added to a transform, this function is applied to the sample in the data grid. If you change your sample or run the job, the computed values for this function are updated. Transforms that change the number of rows in subsequent recipe steps do not affect the values computed for this step.

To perform a simple summing of rows without conditionals, use the `SUM` function. See *SUM Function*.

**Wrangle vs. SQL:** This function is part of Wrangle , a proprietary data transformation language. Wrangle is not SQL. For more information, see *Wrangle Language*.

## Basic Usage

```
sumif(timeoutSecs, errors &gt;= 1)
```

**Output:** Returns the sum of the `timeoutSecs` column when the `errors` value is greater than or equal to 1.

## Syntax and Arguments

```
sumif(col_ref, test_expression) [group:group_col_ref] [limit:limit_count]
```

| Argument | Required? | Data Type | Description |
|---|---|---|---|
| col_ref | Y | string | Reference to the column you wish to evaluate. |
| test_expression | Y | string | Expression that is evaluated. Must resolve to `true` or `false` |

For more information on syntax standards, see *Language Documentation Syntax Notes*.

For more information on the `group` and `limit` parameters, see *Pivot Transform*.

**col_ref**

Name of the column whose values you wish to use in the calculation. Column must be a numeric (Integer or Decimal) type.

**Usage Notes:**

| Required? | Data Type | Example Value |
|---|---|---|

| Yes | String that corresponds to the name of the column | `myValues` |
|---|---|---|

### test_expression

This parameter contains the expression to evaluate. This expression must resolve to a Boolean (`true` or `false`) value.

**Usage Notes:**

| Required? | Data Type | Example Value |
|---|---|---|
| Yes | String expression that evaluates to `true` or `false` | `(LastName == 'Mouse' && FirstName == 'Mickey')` |

## Examples

> **Tip:** For additional examples, see *Common Tasks*.

### Example - Summarize Voter Registrations

This example illustrates how you can use conditional calculation functions.

**Functions**:

| Item | Description |
|---|---|
| SUMIF Function | Generates the sum of rows in each group that meet a specific condition. |
| COUNTDISTINCTIF Function | Generates the count of distinct non-null values for rows in each group that meet a specific condition. |

**Source:**

Here is some example polling data across 16 precincts in 8 cities across 4 counties, where registrations have been invalidated at the polling station, preventing voters from voting. Precincts where this issue has occurred previously have been added to a watch list (`precinctWatchList`).

| totalReg | invalidReg | precinctWatchList | precinctId | cityId | countyId |
|---|---|---|---|---|---|
| 731 | 24 | y | 1 | 1 | 1 |
| 743 | 29 | y | 2 | 1 | 1 |
| 874 | 0 |  | 3 | 2 | 1 |
| 983 | 0 |  | 4 | 2 | 1 |
| 622 | 29 |  | 5 | 3 | 2 |
| 693 | 0 |  | 6 | 3 | 2 |
| 775 | 37 | y | 7 | 4 | 2 |
| 1025 | 49 | y | 8 | 4 | 2 |
| 787 | 13 |  | 9 | 5 | 3 |
| 342 | 0 |  | 10 | 5 | 3 |

| 342 | 39 | y |   | 11 | 6 | 3 |
|-----|-----|-----|---|-----|-----|-----|
| 387 | 28 | y |   | 12 | 6 | 3 |
| 582 | 59 |   |   | 13 | 7 | 4 |
| 244 | 0 |   |   | 14 | 7 | 4 |
| 940 | 6 | y |   | 15 | 8 | 4 |
| 901 | 4 | y |   | 16 | 8 | 4 |

**Transformation:**

First, you want to sum up the invalid registrations (`invalidReg`) for precincts that are already on the watchlist (`precinctWatchList = y`). These sums are grouped by city, which can span multiple precincts:

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | SUMIF(invalidReg, precinctWatchList == "y") |
| Parameter: Group rows by | cityId |
| Parameter: New column name | 'invalidRegbyCityId' |

The `invalidRegbyCityId` column contains invalid registrations across the entire city.

Now, at the county level, you want to identify the number of precincts that were on the watch list and were part of a city-wide registration problem.

In the following step, the number of cities in each count are counted where invalid registrations within a city is greater than `60`.

- This step creates a pivot aggregation.

| Transformation Name | Pivot columns |
|---|---|
| Parameter: Row labels | countyId |
| Parameter: Values | COUNTDISTINCTIF(precinctId, invalidRegbyCityId > 60) |
| Parameter: Max number of columns to create | 1 |

**Results:**

| countyId | countdistinctif_precinctId |
|---|---|
| 1 | 0 |
| 2 | 2 |
| 3 | 2 |
| 4 | 0 |

The voting officials in counties 2 and 3 should investigate their precinct registration issues.

# UNIQUE Function

**Contents:**

---

Extracts the set of unique values from a column into an array stored in a new column. This function is typically part of an aggregation.

> **Tip:** To generate unique values for the generated array, apply the ARRAYUNIQUE function in the next step after this one. See *ARRAYUNIQUE Function*.

Input column can be of any type.

- By default, the list is limited to `1000` values. To change the maximum number of values, specify a value for the `limit` parameter.
- This function is intended to be used as part of an aggregation to return the unique set of values by group. See *Pivot Transform*.

For a version of this function computed over a rolling window of rows, see *ROLLINGLIST Function*.

**Wrangle vs. SQL:** This function is part of Wrangle , a proprietary data transformation language. Wrangle is not SQL. For more information, see *Wrangle Language*.

## Basic Usage

```
unique(Name, 500)
```

**Output:** Returns the unique values for `Name` in an array of all values (up to a count of `500`).

## Syntax and Arguments

```
unique(function_col_ref, [limit_int]) [ group:group_col_ref] [limit:limit_count]
```

| Argument | Required? | Data Type | Description |
|---|---|---|---|
| function_col_ref | Y | string | Name of column to which to apply the function |
| limit_int | N | integer (positive) | Maximum number of unique values to extract into the list array. From `1` to `1000`. |

For more information on the `group` and `limit` parameters, see *Pivot Transform*.

For more information on syntax standards, see *Language Documentation Syntax Notes*.

### function_col_ref

Name of the column from which to extract the list of unique values based on the grouping.

- Literal values are not supported as inputs.
- Multiple columns and wildcards are not supported.

**Usage Notes:**

| Required? | Data Type | Example Value |
|---|---|---|
| Yes | String (column reference) | `myValues` |

### limit_int

Non-negative integer that defines the maximum number of unique values to extract into the list array.

> **NOTE:** If specified, this value must between 1 and 1000, inclusive.

> **NOTE:** Do not use the limiting argument in a `unique` function call on a flat aggregate, in which all values in a column have been inserted into a single cell. In this case, you might be able to use the limit argument if you also specify a `group` parameter. Misuse of the `unique` function can cause the application to crash.

**Usage Notes:**

| Required? | Data Type | Example Value |
|---|---|---|
| No | Integer | `50` |

## Examples

> **Tip:** For additional examples, see *Common Tasks*.

### Example - Colors sold this month

This example demonstrates you to extract values from one column of an array into a new column.

**Functions**:

| Item | Description |
|---|---|
| LIST Function | Extracts the set of values from a column into an array stored in a new column. This function is typically part of an aggregation. |
| UNIQUE Function | Extracts the set of unique values from a column into an array stored in a new column. This function is typically part of an aggregation. |
| DATEFORMAT Function | Formats a specified Datetime set of values according to the specified date format. Source values can be a reference to a column containing Datetime values. |

You have the following set of orders for two months, and you are interested in identifying the set of colors that have been sold for each product for each month and the total quantity of product sold for each month.

**Source:**

| OrderId | Date | Item | Qty | Color |
|---------|---------|-------|-----|--------|
| 1001 | 1/15/15 | Pants | 1 | red |
| 1002 | 1/15/15 | Shirt | 2 | green |
| 1003 | 1/15/15 | Hat | 3 | blue |
| 1004 | 1/16/15 | Shirt | 4 | yellow |
| 1005 | 1/16/15 | Hat | 5 | red |
| 1006 | 1/20/15 | Pants | 6 | green |
| 1007 | 1/15/15 | Hat | 7 | blue |
| 1008 | 4/15/15 | Shirt | 8 | yellow |
| 1009 | 4/15/15 | Shoes | 9 | brown |
| 1010 | 4/16/15 | Pants | 1 | red |
| 1011 | 4/16/15 | Hat | 2 | green |
| 1012 | 4/16/15 | Shirt | 3 | blue |
| 1013 | 4/20/15 | Shoes | 4 | black |
| 1014 | 4/20/15 | Hat | 5 | blue |
| 1015 | 4/20/15 | Pants | 6 | black |

**Transformation:**

To track by month, you need a column containing the month value extracted from the date:

| Transformation Name | `Edit column with formula` |
|---------------------|----------------------------|
| Parameter: Columns | `Date` |
| Parameter: Formula | `DATEFORMAT(Date, 'MMM yyyy')` |

You can use the following transform to check the list of unique values among the colors:

| Transformation Name | `Pivot columns` |
|---------------------|-----------------|
| Parameter: Row labels | `Date` |
| Parameter: Values | `unique(Color, 1000)` |
| Parameter: Max number of columns to create | `10` |

| Date | unique_Color |
|----------|-------------------------------------------------|
| Jan 2015 | ["green","blue","red","yellow"] |
| Apr 2015 | ["brown","blue","red","yellow","black","green"] |

Delete the above transform.

You can aggregate the data in your dataset, grouped by the reformatted `Date` values, and apply the `LIST` function to the `Color` column. In the same aggregation, you can include a summation function for the `Qty` column:

| Transformation Name | Pivot columns |
|---|---|
| Parameter: Row labels | Date |
| Parameter: Values | list(Color, 1000),sum(Qty) |
| Parameter: Max number of columns to create | 10 |

**Results:**

| Date | list_Color | sum_Qty |
|---|---|---|
| Jan 2015 | ["green","blue","blue","red","green","red","yellow"] | 28 |
| Apr 2015 | ["brown","blue","red","yellow","black","blue","black","green"] | 38 |

# VAR Function

Computes the variance among all values in a column. Input column can be of Integer or Decimal. If no numeric values are detected in the input column, the function returns `0`.

The **variance** of a set of values attempts to measure the spread in values around the mean. A variance of zero means that all values are the same, and a small variance means that the values are closely bunched together. A high value for variance indicates that the numbers are spread out widely. Variance is always a positive value.

$Var(X) = [Sum\ ((X - mean(X))^2)]\ /\ Count(X)$

If a row contains a missing or null value, it is not factored into the calculation.

**Relevant terms:**

| Term | Description |
|---|---|
| Population | Population statistical functions are computed from all possible values. See *https://en.wikipedia.org/wiki/Statistical_population*. |
| Sample | Sample-based statistical functions are computed from a subset or sample of all values. See *https://en.wikipedia.org/wiki/Sampling_(statistics)*.<br><br>These function names include `SAMP` in their name.<br><br>> **NOTE:** Statistical sampling has no relationship to the samples taken within the product. When statistical functions are computed during job execution, they are applied across the entire dataset. Sample method calculations are computed at that time. |

- This function is calculated across the entire population.
- For more information on a sampled version of this function, see *VARSAMP Function*.

The square root of variance is standard deviation, which is used to measure variance under the assumption of a bell curve distribution. See *STDEV Function*.

For a version of this function computed over a rolling window of rows, see *ROLLINGVAR Function*.

**Wrangle vs. SQL:** This function is part of Wrangle , a proprietary data transformation language. Wrangle is not SQL. For more information, see *Wrangle Language*.

## Basic Usage

```
var(myRating)
```

**Output:** Returns the variance of the group of values from the `myRating` column.

## Syntax and Arguments

```
var(function_col_ref) [group:group_col_ref] [limit:limit_count]
```

| Argument | Required? | Data Type | Description |
|---|---|---|---|
| function_col_ref | Y | string | Name of column to which to apply the function |

For more information on the `group` and `limit` parameters, see *Pivot Transform*.

For more information on syntax standards, see *Language Documentation Syntax Notes*.

**function_col_ref**

Name of the column the values of which you want to calculate the variance. Column must contain Integer or Decimal values.

- Literal values are not supported as inputs.
- Multiple columns and wildcards are not supported.

**Usage Notes:**

| Required? | Data Type | Example Value |
|---|---|---|
| Yes | String (column reference) | `myValues` |

## Examples

> **Tip:** For additional examples, see *Common Tasks*.

This example illustrates how you can apply statistical functions to your dataset. Calculations include average (mean), max, min, standard deviation, and variance.

**Functions:**

| Item | Description |
|---|---|
| AVERAGE Function | Computes the average (mean) from all row values in a column or group. Input column can be of Integer or Decimal. |
| MIN Function | Computes the minimum value found in all row values in a column. Input column can be of Integer, Decimal or Datetime. |
| MAX Function | Computes the maximum value found in all row values in a column. Inputs can be Integer, Decimal, or Datetime. |
| VAR Function | Computes the variance among all values in a column. Input column can be of Integer or Decimal. If no numeric values are detected in the input column, the function returns $0$. |
| STDEV Function | Computes the standard deviation across all column values of Integer or Decimal type. |
| NUMFORMAT Function | Formats a numeric set of values according to the specified number formatting. Source values can be a literal numeric value, a function returning a numeric value, or reference to a column containing an Integer or Decimal values. |
| MODE Function | Computes the mode (most frequent value) from all row values in a column, according to their grouping. Input column can be of Integer, Decimal, or Datetime type. |

**Source:**

Students took a test and recorded the following scores. You want to perform some statistical analysis on them:

| Student | Score |
|---|---|
| Anna | 84 |

| | |
|---|---|
| Ben | 71 |
| Caleb | 76 |
| Danielle | 87 |
| Evan | 85 |
| Faith | 92 |
| Gabe | 85 |
| Hannah | 99 |
| Ian | 73 |
| Jane | 68 |

**Transformation:**

You can use the following transformations to calculate the average (mean), minimum, and maximum scores:

| Transformation Name | New formula |
|---|---|
| **Parameter: Formula type** | Single row formula |
| **Parameter: Formula** | AVERAGE(Score) |
| **Parameter: New column name** | 'avgScore' |

| Transformation Name | New formula |
|---|---|
| **Parameter: Formula type** | Single row formula |
| **Parameter: Formula** | MIN(Score) |
| **Parameter: New column name** | 'minScore' |

| Transformation Name | New formula |
|---|---|
| **Parameter: Formula type** | Single row formula |
| **Parameter: Formula** | MAX(Score) |
| **Parameter: New column name** | 'maxScore' |

To apply statistical functions to your data, you can use the VAR and STDEV functions, which can be used as the basis for other statistical calculations.

| Transformation Name | New formula |
|---|---|
| **Parameter: Formula type** | Single row formula |
| **Parameter: Formula** | VAR(Score) |
| **Parameter: New column name** | var_Score |

| Transformation Name | New formula |
|---|---|

| Parameter: Formula type | Single row formula |
|---|---|
| Parameter: Formula | STDEV(Score) |
| Parameter: New column name | stdev_Score |

For each score, you can now calculate the variation of each one from the average, using the following:

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | ((Score - avg_Score) / stdev_Score) |
| Parameter: New column name | 'stDevs' |

Now, you want to apply grades based on a formula:

| Grade | standard deviations from avg (stDevs) |
|---|---|
| A | stDevs > 1 |
| B | stDevs > 0.5 |
| C | -1 <= stDevs <= 0.5 |
| D | stDevs < -1 |
| F | stDevs < -2 |

You can build the following transformation using the `IF` function to calculate grades.

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | IF((stDevs > 1),'A',IF((stDevs < -2),'F',IF((stDevs < -1),'D',IF((stDevs > 0.5),'B','C')))) |

To clean up the content, you might want to apply some formatting to the score columns. The following reformats the `stdev_Score` and `stDevs` columns to display two decimal places:

| Transformation Name | Edit column with formula |
|---|---|
| Parameter: Columns | stdev_Score |
| Parameter: Formula | NUMFORMAT(stdev_Score, '##.00') |

| Transformation Name | Edit column with formula |
|---|---|
| Parameter: Columns | stDevs |
| Parameter: Formula | NUMFORMAT(stDevs, '##.00') |

| Transformation Name | New formula |
|---|---|

| Parameter: Formula type | Single row formula |
|---|---|
| Parameter: Formula | MODE(Score) |
| Parameter: New column name | 'modeScore' |

**Results:**

| Student | Score | modeScore | avgScore | minScore | maxScore | var_Score | stdev_Score | stDevs | Grade |
|---|---|---|---|---|---|---|---|---|---|
| Anna | 84 | 85 | 82 | 68 | 99 | 87.000000000 00001 | 9.33 | 0.21 | C |
| Ben | 71 | 85 | 82 | 68 | 99 | 87.000000000 00001 | 9.33 | -1.18 | D |
| Caleb | 76 | 85 | 82 | 68 | 99 | 87.000000000 00001 | 9.33 | -0.64 | C |
| Danielle | 87 | 85 | 82 | 68 | 99 | 87.000000000 00001 | 9.33 | 0.54 | B |
| Evan | 85 | 85 | 82 | 68 | 99 | 87.000000000 00001 | 9.33 | 0.32 | C |
| Faith | 92 | 85 | 82 | 68 | 99 | 87.000000000 00001 | 9.33 | 1.07 | A |
| Gabe | 85 | 85 | 82 | 68 | 99 | 87.000000000 00001 | 9.33 | 0.32 | C |
| Hannah | 99 | 85 | 82 | 68 | 99 | 87.000000000 00001 | 9.33 | 1.82 | A |
| Ian | 73 | 85 | 82 | 68 | 99 | 87.000000000 00001 | 9.33 | -0.96 | C |
| Jane | 68 | 85 | 82 | 68 | 99 | 87.000000000 00001 | 9.33 | -1.50 | D |

# VARIF Function

**Contents:**

- *Basic Usage*
- *Syntax and Arguments*
    - *col_ref*
    - *test_expression*
- *Examples*
    - *Example - Conditional Calculation Functions*

Generates the variance of values by group in a column that meet a specific condition.

> **NOTE:** When added to a transform, this function is applied to the current sample. If you change your sample or run the job, the computed values for this function are updated. Transforms that change the number of rows in subsequent recipe steps do not affect the values computed for this step.

**Relevant terms:**

| Term | Description |
|------|-------------|
| Population | Population statistical functions are computed from all possible values. See *https://en.wikipedia.org/wiki/Statistical_population*. |
| Sample | Sample-based statistical functions are computed from a subset or sample of all values. See *https://en.wikipedia.org/wiki/Sampling_(statistics)*.<br><br>These function names include SAMP in their name.<br><br>> **NOTE:** Statistical sampling has no relationship to the samples taken within the product. When statistical functions are computed during job execution, they are applied across the entire dataset. Sample method calculations are computed at that time. |

- This function is calculated across the entire population.
- For more information on a sampled version of this function, see *VARSAMPIF Function*.

**Wrangle vs. SQL:** This function is part of Wrangle , a proprietary data transformation language. Wrangle is not SQL. For more information, see *Wrangle Language*.

## Basic Usage

```
varif(testScores, ((testScores &gt; 0) &amp;&amp; (testScores &lt; 90))
```

**Output:** Returns the variance of the `testScores` column when the `testScores` value is between 0 and 90.

## Syntax and Arguments

```
varif(col_ref, test_expression) [group:group_col_ref] [limit:limit_count]
```

| Argument | Required? | Data Type | Description |
|----------|-----------|-----------|-------------|

| col_ref | Y | string | Reference to the column you wish to evaluate. |
|---|---|---|---|
| test_expression | Y | string | Expression that is evaluated. Must resolve to `true` or `false` |

For more information on syntax standards, see *Language Documentation Syntax Notes*.

For more information on the `group` and `limit` parameters, see *Pivot Transform*.

**col_ref**

Name of the column whose values you wish to use in the calculation. Column must be a numeric (Integer or Decimal) type.

**Usage Notes:**

| Required? | Data Type | Example Value |
|---|---|---|
| Yes | String that corresponds to the name of the column | `myValues` |

**test_expression**

This parameter contains the expression to evaluate. This expression must resolve to a Boolean (`true` or `false`) value.

**Usage Notes:**

| Required? | Data Type | Example Value |
|---|---|---|
| Yes | String expression that evaluates to `true` or `false` | `(LastName == 'Mouse' && FirstName == 'Mickey')` |

Examples

> **Tip:** For additional examples, see *Common Tasks*.

**Example - Conditional Calculation Functions**

This example illustrates how to use the conditional calculation functions.

**Functions**:

| Item | Description |
|---|---|
| AVERAGEIF Function | Generates the average value of rows in each group that meet a specific condition. Generated value is of Decimal type. |
| MINIF Function | Generates the minimum value of rows in each group that meet a specific condition. Inputs can be Integer, Decimal, or Datetime. |
| MAXIF Function | Generates the maximum value of rows in each group that meet a specific condition. Inputs can be Integer, Decimal, or Datetime. |
| VARIF Function | Generates the variance of values by group in a column that meet a specific condition. |
| STDEVIF Function | Generates the standard deviation of values by group in a column that meet a specific condition. |

**Source:**

Here is some example weather data:

| date | city | rain | temp | wind |
|------|------|------|------|------|
| 1/23/17 | Valleyville | 0.00 | 12.8 | 6.7 |
| 1/23/17 | Center Town | 0.31 | 9.4 | 5.3 |
| 1/23/17 | Magic Mountain | 0.00 | 0.0 | 7.3 |
| 1/24/17 | Valleyville | 0.25 | 17.2 | 3.3 |
| 1/24/17 | Center Town | 0.54 | 1.1 | 7.6 |
| 1/24/17 | Magic Mountain | 0.32 | 5.0 | 8.8 |
| 1/25/17 | Valleyville | 0.02 | 3.3 | 6.8 |
| 1/25/17 | Center Town | 0.83 | 3.3 | 5.1 |
| 1/25/17 | Magic Mountain | 0.59 | -1.7 | 6.4 |
| 1/26/17 | Valleyville | 1.08 | 15.0 | 4.2 |
| 1/26/17 | Center Town | 0.96 | 6.1 | 7.6 |
| 1/26/17 | Magic Mountain | 0.77 | -3.9 | 3.0 |
| 1/27/17 | Valleyville | 1.00 | 7.2 | 2.8 |
| 1/27/17 | Center Town | 1.32 | 20.0 | 0.2 |
| 1/27/17 | Magic Mountain | 0.77 | 5.6 | 5.2 |
| 1/28/17 | Valleyville | 0.12 | -6.1 | 5.1 |
| 1/28/17 | Center Town | 0.14 | 5.0 | 4.9 |
| 1/28/17 | Magic Mountain | 1.50 | 1.1 | 0.4 |
| 1/29/17 | Valleyville | 0.36 | 13.3 | 7.3 |
| 1/29/17 | Center Town | 0.75 | 6.1 | 9.0 |
| 1/29/17 | Magic Mountain | 0.60 | 3.3 | 6.0 |

**Transformation:**

The following computes average temperature for rainy days by city:

| | |
|---|---|
| **Transformation Name** | `New formula` |
| **Parameter: Formula type** | `Single row formula` |
| **Parameter: Formula** | `AVERAGEIF(temp, rain > 0)` |
| **Parameter: Group rows by** | `city` |
| **Parameter: New column name** | `'avgTempWRain'` |

The following computes maximum wind for sub-zero days by city:

| | |
|---|---|
| **Transformation Name** | `New formula` |
| **Parameter: Formula type** | `Single row formula` |
| **Parameter: Formula** | `MAXIF(wind,temp < 0)` |

| Parameter: Group rows by | city |
|---|---|
| Parameter: New column name | 'maxWindSubZero' |

This step calculates the minimum temp when the wind is less than 5 mph by city:

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | MINIF(temp,wind<5) |
| Parameter: Group rows by | city |
| Parameter: New column name | 'minTempWind5' |

This step computes the variance in temperature for rainy days by city:

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | VARIF(temp,rain >0) |
| Parameter: Group rows by | city |
| Parameter: New column name | 'varTempWRain' |

The following computes the standard deviation in rainfall for Center Town:

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | STDEVIF(rain,city=='Center Town') |
| Parameter: Group rows by | city |
| Parameter: New column name | 'stDevRainCT' |

You can use the following transforms to format the generated output. Note the $col placeholder value for the multi-column transforms:

| Transformation Name | Edit column with formula |
|---|---|
| Parameter: Columns | stDevRainCenterTown,maxWindSubZero |
| Parameter: Formula | numformat($col,'##.##') |

Since the following rely on data that has only one significant digit, you should format them differently:

| Transformation Name | Edit column with formula |
|---|---|
| Parameter: Columns | varTempWRain,avgTempWRain,minTempWind5 |

| | Parameter: Formula | `numformat($col,'##.#')` |
| --- | --- | --- |

## Results:

| date | city | rain | temp | wind | avgTempWRain | maxWindSubZero | minTempWind5 | varTempWRain | stDevRain |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 1/23/17 | Valleyville | 0.00 | 12.8 | 6.7 | 8.3 | 5.1 | 7.2 | 63.8 | 0.37 |
| 1/23/17 | Center Town | 0.31 | 9.4 | 5.3 | 7.3 | | 5 | 32.6 | 0.37 |
| 1/23/17 | Magic Mountain | 0.00 | 0.0 | 7.3 | 1.6 | 6.43 | -3.9 | 12 | 0.37 |
| 1/24/17 | Valleyville | 0.25 | 17.2 | 3.3 | 8.3 | 5.1 | 7.2 | 63.8 | 0.37 |
| 1/24/17 | Center Town | 0.54 | 1.1 | 7.6 | 7.3 | | 5 | 32.6 | 0.37 |
| 1/24/17 | Magic Mountain | 0.32 | 5.0 | 8.8 | 1.6 | 6.43 | -3.9 | 12 | 0.37 |
| 1/25/17 | Valleyville | 0.02 | 3.3 | 6.8 | 8.3 | 5.1 | 7.2 | 63.8 | 0.37 |
| 1/25/17 | Center Town | 0.83 | 3.3 | 5.1 | 7.3 | | 5 | 32.6 | 0.37 |
| 1/25/17 | Magic Mountain | 0.59 | -1.7 | 6.4 | 1.6 | 6.43 | -3.9 | 12 | 0.37 |
| 1/26/17 | Valleyville | 1.08 | 15.0 | 4.2 | 8.3 | 5.1 | 7.2 | 63.8 | 0.37 |
| 1/26/17 | Center Town | 0.96 | 6.1 | 7.6 | 7.3 | | 5 | 32.6 | 0.37 |
| 1/26/17 | Magic Mountain | 0.77 | -3.9 | 3.0 | 1.6 | 6.43 | -3.9 | 12 | 0.37 |
| 1/27/17 | Valleyville | 1.00 | 7.2 | 2.8 | 8.3 | 5.1 | 7.2 | 63.8 | 0.37 |
| 1/27/17 | Center Town | 1.32 | 20.0 | 0.2 | 7.3 | | 5 | 32.6 | 0.37 |
| 1/27/17 | Magic Mountain | 0.77 | 5.6 | 5.2 | 1.6 | 6.43 | -3.9 | 12 | 0.37 |
| 1/28/17 | Valleyville | 0.12 | -6.1 | 5.1 | 8.3 | 5.1 | 7.2 | 63.8 | 0.37 |
| 1/28/17 | Center Town | 0.14 | 5.0 | 4.9 | 7.3 | | 5 | 32.6 | 0.37 |
| 1/28/17 | Magic Mountain | 1.50 | 1.1 | 0.4 | 1.6 | 6.43 | -3.9 | 12 | 0.37 |
| 1/29/17 | Valleyville | 0.36 | 13.3 | 7.3 | 8.3 | 5.1 | 7.2 | 63.8 | 0.37 |

| 1/29/17 | Center Town | 0.75 | 6.1 | 9.0 | 7.3 | | 5 | 32.6 | 0.37 |
| 1/29/17 | Magic Mountain | 0.60 | 3.3 | 6.0 | 1.6 | 6.43 | -3.9 | 12 | 0.37 |

# VARSAMP Function

Computes the variance among all values in a column using the sample statistical method. Input column can be of Integer or Decimal. If no numeric values are detected in the input column, the function returns `0`.

The **variance** of a set of values attempts to measure the spread in values around the mean. A variance of zero means that all values are the same, and a small variance means that the values are closely bunched together. A high value for variance indicates that the numbers are spread out widely. Variance is always a positive value.

If a row contains a missing or null value, it is not factored into the calculation.

> **NOTE:** This function applies to a sample of the entire population. More information is below.

**Relevant terms:**

| Term | Description |
|------|-------------|
| Population | Population statistical functions are computed from all possible values. See *https://en.wikipedia.org/wiki/Statistical_population*. |
| Sample | Sample-based statistical functions are computed from a subset or sample of all values. See *https://en.wikipedia.org/wiki/Sampling_(statistics)*.<br><br>These function names include $\mathrm{SAMP}$ in their name.<br><br>> **NOTE:** Statistical sampling has no relationship to the samples taken within the product. When statistical functions are computed during job execution, they are applied across the entire dataset. Sample method calculations are computed at that time. |

- This function is calculated across a sample of all values.
- For more information on a population version of this function, see *VAR Function*.

In the following computation, the sample method computes variances with *N - 1* as the divisor.

*Var(X) = [Sum ((X - mean(X))$^2$)] / (Count(X) - 1)*

The square root of variance is standard deviation, which is used to measure variance under the assumption of a bell curve distribution. See *STDEV Function*.

For a version of this function computed over a rolling window of rows, see *ROLLINGVAR Function*.

**Wrangle vs. SQL:** This function is part of Wrangle , a proprietary data transformation language. Wrangle is not SQL. For more information, see *Wrangle Language*.

## Basic Usage

```
varsamp(myRating)
```

**Output:** Returns the variance of the group of values from the `myRating` column using the sample method of calculation.

## Syntax and Arguments

```
<span>varsamp</span>(col_ref) [group:group_col_ref] [limit:limit_count]
```

| Argument | Required? | Data Type | Description |
|---|---|---|---|
| function_col_ref | Y | string | Name of column to which to apply the function |

For more information on the `group` and `limit` parameters, see *Pivot Transform*.

For more information on syntax standards, see *Language Documentation Syntax Notes*.

**col_ref**

Name of the column whose values you wish to use in the calculation. Column must be a numeric (Integer or Decimal) type.

- Literal values are not supported as inputs.
- Multiple columns and wildcards are not supported.

**Usage Notes:**

| Required? | Data Type | Example Value |
|---|---|---|
| Yes | String (column reference) | `myValues` |

## Examples

> **Tip:** For additional examples, see *Common Tasks*.

This example shows some of the statistical functions that use the sample method of computation.

**Functions**:

| Item | Description |
|---|---|
| STDEVSAMP Function | Computes the standard deviation across column values of Integer or Decimal type using the sample statistical method. |
| VARSAMP Function | Computes the variance among all values in a column using the sample statistical method. Input column can be of Integer or Decimal. If no numeric values are detected in the input column, the function returns $0$. |
| STDEVSAMP IF Function | Generates the standard deviation of values by group in a column that meet a specific condition using the sample statistical method. |
| VARSAMPIF Function | Generates the variance of values by group in a column that meet a specific condition using the sample statistical method. |
| ROUND Function | Rounds input value to the nearest integer. Input can be an Integer, a Decimal, a column reference, or an expression. Optional second argument can be used to specify the number of digits to which to round. |

**Source:**

Students took tests on three consecutive Saturdays:

| Student | Date | Score |
|---------|------|-------|
| Andrew | 11/9/19 | 81 |
| Bella | 11/9/19 | 84 |
| Christina | 11/9/19 | 79 |
| David | 11/9/19 | 64 |
| Ellen | 11/9/19 | 61 |
| Fred | 11/9/19 | 63 |
| Andrew | 11/16/19 | 73 |
| Bella | 11/16/19 | 88 |
| Christina | 11/16/19 | 78 |
| David | 11/16/19 | 67 |
| Ellen | 11/16/19 | 87 |
| Fred | 11/16/19 | 90 |
| Andrew | 11/23/19 | 76 |
| Bella | 11/23/19 | 93 |
| Christina | 11/23/19 | 81 |
| David | 11/23/19 | 97 |
| Ellen | 11/23/19 | 97 |
| Fred | 11/23/19 | 91 |

**Transformation:**

You can use the following transformations to calculate standard deviation and variance across all dates using the sample method. Each computation has been rounded to three digits.

| Transformation Name | New formula |
|---------------------|-------------|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | round(stdevsamp(Score), 3) |
| Parameter: New column name | 'stdevSamp' |

| Transformation Name | New formula |
|---------------------|-------------|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | round(varsamp(Score), 3) |
| Parameter: New column name | 'varSamp' |

You can use the following to limit the previous statistical computations to the last two Saturdays of testing:

| Transformation Name | New formula |
|---------------------|-------------|
| Parameter: Formula type | Single row formula |

| Parameter: Formula | round(stdevsampif(Score, Date != '11\/9\/2019'), 3) |
|---|---|
| Parameter: New column name | 'stdevSampIf' |

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | round(varsampif(Score, Date != '11\/9\/2019'), 3) |
| Parameter: New column name | 'varSampIf' |

**Results:**

| Student | Date | Score | varSampIf | stdevSampIf | varSamp | stdevSamp |
|---|---|---|---|---|---|---|
| Andrew | 11/9/19 | 81 | 94.515 | 9.722 | 131.673 | 11.475 |
| Bella | 11/9/19 | 84 | 94.515 | 9.722 | 131.673 | 11.475 |
| Christina | 11/9/19 | 79 | 94.515 | 9.722 | 131.673 | 11.475 |
| David | 11/9/19 | 64 | 94.515 | 9.722 | 131.673 | 11.475 |
| Ellen | 11/9/19 | 61 | 94.515 | 9.722 | 131.673 | 11.475 |
| Fred | 11/9/19 | 63 | 94.515 | 9.722 | 131.673 | 11.475 |
| Andrew | 11/16/19 | 73 | 94.515 | 9.722 | 131.673 | 11.475 |
| Bella | 11/16/19 | 88 | 94.515 | 9.722 | 131.673 | 11.475 |
| Christina | 11/16/19 | 78 | 94.515 | 9.722 | 131.673 | 11.475 |
| David | 11/16/19 | 67 | 94.515 | 9.722 | 131.673 | 11.475 |
| Ellen | 11/16/19 | 87 | 94.515 | 9.722 | 131.673 | 11.475 |
| Fred | 11/16/19 | 90 | 94.515 | 9.722 | 131.673 | 11.475 |
| Andrew | 11/23/19 | 76 | 94.515 | 9.722 | 131.673 | 11.475 |
| Bella | 11/23/19 | 93 | 94.515 | 9.722 | 131.673 | 11.475 |
| Christina | 11/23/19 | 81 | 94.515 | 9.722 | 131.673 | 11.475 |
| David | 11/23/19 | 97 | 94.515 | 9.722 | 131.673 | 11.475 |
| Ellen | 11/23/19 | 97 | 94.515 | 9.722 | 131.673 | 11.475 |
| Fred | 11/23/19 | 91 | 94.515 | 9.722 | 131.673 | 11.475 |

# VARSAMPIF Function

**Contents:**

Generates the variance of values by group in a column that meet a specific condition using the sample statistical method.

> **NOTE:** When added to a transform, this function is applied to the current sample. If you change your sample or run the job, the computed values for this function are updated. Transforms that change the number of rows in subsequent recipe steps do not affect the values computed for this step.

> **NOTE:** This function applies to a sample of the entire population. More information is below.

**Relevant terms:**

| Term | Description |
| --- | --- |
| Population | Population statistical functions are computed from all possible values. See *https://en.wikipedia.org/wiki/Statistical_population*. |
| Sample | Sample-based statistical functions are computed from a subset or sample of all values. See *https://en.wikipedia.org/wiki/Sampling_(statistics)*.<br><br>These function names include $\mathrm{SAMP}$ in their name.<br><br>> **NOTE:** Statistical sampling has no relationship to the samples taken within the product. When statistical functions are computed during job execution, they are applied across the entire dataset. Sample method calculations are computed at that time. |

- This function is calculated across a sample of all values.
- For more information on a population version of this function, see *VARIF Function*.

**Wrangle vs. SQL:** This function is part of Wrangle , a proprietary data transformation language. Wrangle is not SQL. For more information, see *Wrangle Language*.

## Basic Usage

```
varsampif(testScores, ((testScores &gt; 0) &amp;&amp; (testScores &lt; 90)))
```

**Output:** Returns the variance of the `testScores` column when the `testScores` value is between 0 and 90 using the sample method of calculation.

## Syntax and Arguments

```
<span>varsampif</span>(col_ref, test_expression) [group:group_col_ref] [limit:limit_count]
```

| Argument | Required? | Data Type | Description |
|---|---|---|---|
| col_ref | Y | string | Reference to the column you wish to evaluate. |
| test_expression | Y | string | Expression that is evaluated. Must resolve to `true` or `false` |

For more information on syntax standards, see *Language Documentation Syntax Notes*.

For more information on the `group` and `limit` parameters, see *Pivot Transform*.

**col_ref**

Name of the column whose values you wish to use in the calculation. Column must be a numeric (Integer or Decimal) type.

**Usage Notes:**

| Required? | Data Type | Example Value |
|---|---|---|
| Yes | String that corresponds to the name of the column | `myValues` |

**test_expression**

This parameter contains the expression to evaluate. This expression must resolve to a Boolean (`true` or `false`) value.

**Usage Notes:**

| Required? | Data Type | Example Value |
|---|---|---|
| Yes | String expression that evaluates to `true` or `false` | `(LastName == 'Mouse' && FirstName == 'Mickey')` |

Examples

> **Tip:** For additional examples, see *Common Tasks*.

**Example - Conditional Calculation Functions**

This example shows some of the statistical functions that use the sample method of computation.

**Functions**:

| Item | Description |
|---|---|
| STDEVSAMP Function | Computes the standard deviation across column values of Integer or Decimal type using the sample statistical method. |
| VARSAMP Function | Computes the variance among all values in a column using the sample statistical method. Input column can be of Integer or Decimal. If no numeric values are detected in the input column, the function returns $0$. |

| | |
|---|---|
| STDEVSAMP IF Function | Generates the standard deviation of values by group in a column that meet a specific condition using the sample statistical method. |
| VARSAMPIF Function | Generates the variance of values by group in a column that meet a specific condition using the sample statistical method. |
| ROUND Function | Rounds input value to the nearest integer. Input can be an Integer, a Decimal, a column reference, or an expression. Optional second argument can be used to specify the number of digits to which to round. |

**Source:**

Students took tests on three consecutive Saturdays:

| Student | Date | Score |
|---|---|---|
| Andrew | 11/9/19 | 81 |
| Bella | 11/9/19 | 84 |
| Christina | 11/9/19 | 79 |
| David | 11/9/19 | 64 |
| Ellen | 11/9/19 | 61 |
| Fred | 11/9/19 | 63 |
| Andrew | 11/16/19 | 73 |
| Bella | 11/16/19 | 88 |
| Christina | 11/16/19 | 78 |
| David | 11/16/19 | 67 |
| Ellen | 11/16/19 | 87 |
| Fred | 11/16/19 | 90 |
| Andrew | 11/23/19 | 76 |
| Bella | 11/23/19 | 93 |
| Christina | 11/23/19 | 81 |
| David | 11/23/19 | 97 |
| Ellen | 11/23/19 | 97 |
| Fred | 11/23/19 | 91 |

**Transformation:**

You can use the following transformations to calculate standard deviation and variance across all dates using the sample method. Each computation has been rounded to three digits.

| | |
|---|---|
| **Transformation Name** | `New formula` |
| **Parameter: Formula type** | `Single row formula` |
| **Parameter: Formula** | `round(stdevsamp(Score), 3)` |
| **Parameter: New column name** | `'stdevSamp'` |

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | round(varsamp(Score), 3) |
| Parameter: New column name | 'varSamp' |

You can use the following to limit the previous statistical computations to the last two Saturdays of testing:

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | round(stdevsampif(Score, Date != '11\/9\/2019'), 3) |
| Parameter: New column name | 'stdevSampIf' |

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | round(varsampif(Score, Date != '11\/9\/2019'), 3) |
| Parameter: New column name | 'varSampIf' |

**Results:**

| Student | Date | Score | varSampIf | stdevSampIf | varSamp | stdevSamp |
|---|---|---|---|---|---|---|
| Andrew | 11/9/19 | 81 | 94.515 | 9.722 | 131.673 | 11.475 |
| Bella | 11/9/19 | 84 | 94.515 | 9.722 | 131.673 | 11.475 |
| Christina | 11/9/19 | 79 | 94.515 | 9.722 | 131.673 | 11.475 |
| David | 11/9/19 | 64 | 94.515 | 9.722 | 131.673 | 11.475 |
| Ellen | 11/9/19 | 61 | 94.515 | 9.722 | 131.673 | 11.475 |
| Fred | 11/9/19 | 63 | 94.515 | 9.722 | 131.673 | 11.475 |
| Andrew | 11/16/19 | 73 | 94.515 | 9.722 | 131.673 | 11.475 |
| Bella | 11/16/19 | 88 | 94.515 | 9.722 | 131.673 | 11.475 |
| Christina | 11/16/19 | 78 | 94.515 | 9.722 | 131.673 | 11.475 |
| David | 11/16/19 | 67 | 94.515 | 9.722 | 131.673 | 11.475 |
| Ellen | 11/16/19 | 87 | 94.515 | 9.722 | 131.673 | 11.475 |
| Fred | 11/16/19 | 90 | 94.515 | 9.722 | 131.673 | 11.475 |
| Andrew | 11/23/19 | 76 | 94.515 | 9.722 | 131.673 | 11.475 |
| Bella | 11/23/19 | 93 | 94.515 | 9.722 | 131.673 | 11.475 |
| Christina | 11/23/19 | 81 | 94.515 | 9.722 | 131.673 | 11.475 |
| David | 11/23/19 | 97 | 94.515 | 9.722 | 131.673 | 11.475 |
| Ellen | 11/23/19 | 97 | 94.515 | 9.722 | 131.673 | 11.475 |

| Fred | 11/23/19 | 91 | 94.515 | 9.722 | 131.673 | 11.475 |

# Logical Functions

This category encompasses standard logical operators and their corresponding functions: `AND`, `OR`, and `NOT`.

# Logical Operators

**Contents:**

Logical operators (and, or, not) enable you to logically combine multiple expressions to evaluate a larger, more complex expression whose output is `true` or `false`.

```
(left-hand side) (operator) (right-hand side)
```

These evaluations result in a Boolean output. The following operators are supported:

| Operator Name | Symbol | Example Expression | Output | Notes |
|---|---|---|---|---|
| and | && | `((1 == 1) && (2 == 2))` | `true` | |
| | | `((1 == 1) && (2 == 3))` | `false` | |
| or | \|\| | `((1 == 1) \|\| (2 == 2))` | `true` | Exclusive or (xor) is not supported. |
| | | `((1 == 2) \|\| (2 == 3))` | `false` | |
| not | ! | `!(1 == 1)` | `false` | |
| | | `!(1 == 2)` | `true` | |

The above examples apply to integer values only. Below, you can review how the comparison operators apply to different data types.

## Usage

Logical operators are used to perform evaluations of expressions covering a variety of data types. Typically, they are applied in evaluations of values or rows.

Example data:

| X | Y |
|---|---|
| true | true |
| true | false |
| false | true |
| false | false |

**Transforms:**

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |

| Parameter: Formula | (X && Y) |
| --- | --- |
| Parameter: New column name | 'col_and' |

| Transformation Name | New formula |
| --- | --- |
| Parameter: Formula type | Single row formula |
| Parameter: Formula | (X \|\| Y) |
| Parameter: New column name | 'col_or' |

| Transformation Name | New formula |
| --- | --- |
| Parameter: Formula type | Single row formula |
| Parameter: Formula | !(or) |
| Parameter: New column name | 'col_not_and' |

| Transformation Name | New formula |
| --- | --- |
| Parameter: Formula type | Single row formula |
| Parameter: Formula | !(or) |
| Parameter: New column name | 'col_not_or' |

**Results:**

Your output looks like the following:

| X | Y | col_and | col_or | col_not_and | col_not_or |
| --- | --- | --- | --- | --- | --- |
| true | true | true | true | false | false |
| true | false | false | true | true | false |
| false | true | false | true | true | false |
| false | false | false | false | true | true |

## Examples

> **Tip:** For additional examples, see *Common Tasks*.

**and**

| Column Type | Example Transform | Output | Notes |
| --- | --- | --- | --- |

| Integer /Decimal | Transformation Name | Edit column with formula | Set the value of the `InRange` column to `true` if the value of the `Input` column is between 10 and 90, inclusive.<br>Otherwise, `InRange` column is `false`. | |
| --- | --- | --- | --- | --- |
| | Parameter: Columns | `InRange` | | |
| | Parameter: Formula | `((Input >= 10) && (Input <= 90))` | | |

| Datetime | Transformation Name | Filter rows | Delete all rows in which the `Date` value falls somewhere in 2014. | |
| --- | --- | --- | --- | --- |
| | Parameter: Condition | `Custom formula` | | |
| | Parameter: Type of formula | `Custom single` | | |
| | Parameter: Condition | `((Date >= DATE (2014, 01, 01)) && (Date <= DATE (2014, 12, 31))` | | |
| | Parameter: Action | `Delete matching rows` | | |

| String | Transformation Name | New formula | For U.S. State names, the generated column contains `true` for the following values:<br><br>• `Alabama`<br>• `Alaska`<br>• `Arizona`<br><br>For all other values, the generated value is `false`. | • See *LEFT Function* .<br>• See *RIGHT Function* . |
| --- | --- | --- | --- | --- |
| | Parameter: Formula type | `Single row formula` | | |
| | Parameter: Formula | `((LEFT(USStates, 1) == "A") && (RIGHT(USStates, 1) == "A"))` | | |

**or**

| Column Type | Example Transform | | Output | Notes |
| --- | --- | --- | --- | --- |
| Integer /Decimal | Transformation Name | Edit column with formula | • In the `BigOrder` column, set the value to `true` if the value of `Total` is more than 1,000,000 or the value of `Qty` is more than 1000.<br>• Otherwise, the value is `false`. | |
| | Parameter: Columns | `BigOrder` | | |
| | Parameter: Formula | `((Total > 1000000) \|\| (Qty > 1000))` | | |

| | | | | |
|---|---|---|---|---|
| Datetime | **Transformation Name** | Filter rows | Delete all rows in the dataset where the `Date` value is earlier than 01/01/1950 or later than 12/31/2050. | |
| | **Parameter: Condition** | Custom formula | | |
| | **Parameter: Type of formula** | Custom single | | |
| | **Parameter: Condition** | ((Date <= DATE (1950, 01, 01)) \|\| (Date >= DATE (2050, 12, 31)) | | |
| | **Parameter: Action** | Delete matching rows | | |
| String | **Transformation Name** | New formula | • Generate the new `good_car` column containing `true` if the `Brand` is `subaru` or the `Color` is `green`.<br>• Otherwise, the `good_car` value is `false`. | |
| | **Parameter: Formula type** | Single row formula | | |
| | **Parameter: Formula** | ((Brand == 'subaru') \|\| ('Color' == 'green')) | | |
| | **Parameter: New column name** | 'good_car' | | |

## not

| Column Type | Example Transform | | Output | Notes |
|---|---|---|---|---|
| Integer /Decimal | **Transformation Name** | Filter rows | Keep all rows for houses that do not meet any of these criteria:<br><br>• smaller than 1300 square feet,<br>• less than 2 bathrooms,<br>• less than 2.5 bedrooms. | |
| | **Parameter: Condition** | Custom formula | | |
| | **Parameter: Type of formula** | Custom single | | |
| | **Parameter: Condition** | !((sqft < 1300) && (bath < 2) && (bed < 2.5)) | | |
| | **Parameter: Action** | Keep matching rows | | |

| Datetime | | | Keep all rows in the dataset where the year of the `Date` value is not 2016. | |
|---|---|---|---|---|
| | **Transformation Name** | `Filter rows` | | |
| | **Parameter: Condition** | `Custom formula` | | |
| | **Parameter: Type of formula** | `Custom single` | | |
| | **Parameter: Condition** | `!(YEAR(Date) == '2016')` | | |
| | **Parameter: Action** | `Keep matching rows` | | |
| String | | | Delete all rows in which the value of the `status` column is not `Keep_It`. | |
| | **Transformation Name** | `Filter rows` | | |
| | **Parameter: Condition** | `Custom formula` | | |
| | **Parameter: Type of formula** | `Custom single` | | |
| | **Parameter: Condition** | `!(status == 'Keep_It')` | | |
| | **Parameter: Action** | `Delete matching rows` | | |

# AND Function

Returns `true` if both arguments evaluate to `true`. Equivalent to the `&&` operator.

- Each argument can be a literal Boolean, a function returning a Boolean, or a reference to a column containing Boolean values.

Since the `AND` function returns a Boolean value, it can be used as a function or a conditional.

> **NOTE:** Within an expression, you might choose to use the corresponding operator, instead of this function. For more information, see *Logical Operators*.

**Wrangle vs. SQL:** This function is part of Wrangle , a proprietary data transformation language. Wrangle is not SQL. For more information, see *Wrangle Language*.

## Basic Usage

```
and(finalScoreEnglish &gt;= 60, finalScoreMath &gt;=60)
```

**Output:** Returns `true` if the values in the `finalScoreEnglish` and `finalScoreMath` columns are greater than or equal to `60`. Otherwise, the value is `false`.

## Syntax and Arguments

```
and(value1, value2)
```

| Argument | Required? | Data Type | Description |
|----------|-----------|-----------|-------------|
| value1 | Y | string | The first value must be a Boolean literal, column reference, or expression that evaluates to `true` or `false`. |
| value2 | Y | string | The first value must be a Boolean literal, column reference, or expression that evaluates to `true` or `false`. |

For more information on syntax standards, see *Language Documentation Syntax Notes*.

**value1, value2**

Expressions, column references or literals to compare as Boolean values.

- Missing or mismatched values generate missing results.

**Usage Notes:**

| Required? | Data Type | Example Value |
|-----------|-----------|---------------|
| Yes | Function or column reference returning a Boolean value or Boolean literal | `myHeight > 2.00` |

# Examples

> **Tip:** For additional examples, see *Common Tasks*.

### Example - Logical Functions

This example demonstrate the AND, OR, and NOT logical functions.

In this example, the dataset contains results from survey data on two questions about customers. The yes/no answers to each question determine if the customer is 1) still active, and 2) interested in a new offering.

**Functions:**

| Item | Description |
|------|-------------|
| AND Function | Returns `true` if both arguments evaluate to `true`. Equivalent to the `&&` operator. |
| OR Function | Returns `true` if either argument evaluates to `true`. Equivalent to the `||` operator. |
| NOT Function | Returns `true` if the argument evaluates to `false`, and vice-versa. Equivalent to the `!` operator. |

**Source:**

| Customer | isActive | isInterested |
|----------|----------|--------------|
| CustA | Y | Y |
| CustB | Y | N |
| CustC | N | Y |
| CustD | N | N |

**Transformation:**

Customers that are both active and interested should receive a phone call:

| Transformation Name | `New formula` |
|---------------------|---------------|
| **Parameter: Formula type** | `Single row formula` |
| **Parameter: Formula** | `AND(isActive, isInterested)` |
| **Parameter: New column name** | `'phoneCall'` |

Customers that are either active or interested should receive an email:

| Transformation Name | `New formula` |
|---------------------|---------------|
| **Parameter: Formula type** | `Single row formula` |
| **Parameter: Formula** | `OR(isActive, isInterested)` |
| **Parameter: New column name** | `'sendEmail'` |

Customers that are neither active or interested should be dropped from consideration for the offering:

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | AND(NOT(isActive),NOT(isInterested) |
| Parameter: New column name | 'dropCust' |

A savvy marketer might decide that if a customer receives a phone call, that customer should not be bothered with an email, as well:

| Transformation Name | Edit column with formula |
|---|---|
| Parameter: Columns | sendEmail |
| Parameter: Formula | IF(phoneCall == "TRUE", FALSE, sendEmail) |

**Results:**

| Customer | isActive | isInterested | dropCust | sendEmail | phoneCall |
|---|---|---|---|---|---|
| CustA | Y | Y | FALSE | FALSE | TRUE |
| CustB | Y | N | FALSE | TRUE | FALSE |
| CustC | N | Y | FALSE | TRUE | FALSE |
| CustD | N | N | TRUE | FALSE | FALSE |

# OR Function

Returns `true` if either argument evaluates to `true`. Equivalent to the `||` operator.

- Each argument can be a literal Boolean, a function returning a Boolean, or a reference to a column containing Boolean values.

Since the function returns a Boolean value, it can be used as a function or a conditional.

> **NOTE:** Within an expression, you might choose to use the corresponding operator, instead of this function. For more information, see *Logical Operators*.

**Wrangle vs. SQL:** This function is part of Wrangle , a proprietary data transformation language. Wrangle is not SQL. For more information, see *Wrangle Language*.

## Basic Usage

```
or(daysBillOverdue &gt; 90, violationsCount &gt; 2)
```

**Output:** If the value in the `daysBillOverdue` column is greater than `90` or the value in `violationsCount` column is greater than `2`, then the returned value is `true`. Otherwise, the value is `false`.

## Syntax and Arguments

```
or(value1, value2)
```

| Argument | Required? | Data Type | Description |
|----------|-----------|-----------|-------------|
| value1 | Y | string | The first value must be a Boolean literal, column reference, or expression that evaluates to `true` or `false`. |
| value2 | Y | string | The first value must be a Boolean literal, column reference, or expression that evaluates to `true` or `false`. |

For more information on syntax standards, see *Language Documentation Syntax Notes*.

**value1, value2**

Expressions, column references or literals to compare as Boolean values.

- Missing or mismatched values generate missing results.

**Usage Notes:**

| Required? | Data Type | Example Value |
|-----------|-----------|---------------|
| Yes | Function or column reference returning a Boolean value or Boolean literal | `myHeight > 2.00` |

# Examples

> **Tip:** For additional examples, see *Common Tasks*.

## Example - Logical Functions

This example demonstrate the `AND`, `OR`, and `NOT` logical functions.

In this example, the dataset contains results from survey data on two questions about customers. The yes/no answers to each question determine if the customer is 1) still active, and 2) interested in a new offering.

**Functions:**

| Item | Description |
|---|---|
| AND Function | Returns `true` if both arguments evaluate to `true`. Equivalent to the `&&` operator. |
| OR Function | Returns `true` if either argument evaluates to `true`. Equivalent to the `││` operator. |
| NOT Function | Returns `true` if the argument evaluates to `false`, and vice-versa. Equivalent to the `!` operator. |

**Source:**

| Customer | isActive | isInterested |
|---|---|---|
| CustA | Y | Y |
| CustB | Y | N |
| CustC | N | Y |
| CustD | N | N |

**Transformation:**

Customers that are both active and interested should receive a phone call:

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | AND(isActive, isInterested) |
| Parameter: New column name | 'phoneCall' |

Customers that are either active or interested should receive an email:

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | OR(isActive, isInterested) |
| Parameter: New column name | 'sendEmail' |

Customers that are neither active or interested should be dropped from consideration for the offering:

| Transformation Name | New formula |
|---|---|
| **Parameter: Formula type** | Single row formula |
| **Parameter: Formula** | AND(NOT(isActive),NOT(isInterested) |
| **Parameter: New column name** | 'dropCust' |

A savvy marketer might decide that if a customer receives a phone call, that customer should not be bothered with an email, as well:

| Transformation Name | Edit column with formula |
|---|---|
| **Parameter: Columns** | sendEmail |
| **Parameter: Formula** | IF(phoneCall == "TRUE", FALSE, sendEmail) |

### Results:

| Customer | isActive | isInterested | dropCust | sendEmail | phoneCall |
|---|---|---|---|---|---|
| CustA | Y | Y | FALSE | FALSE | TRUE |
| CustB | Y | N | FALSE | TRUE | FALSE |
| CustC | N | Y | FALSE | TRUE | FALSE |
| CustD | N | N | TRUE | FALSE | FALSE |

# NOT Function

Returns `true` if the argument evaluates to `false`, and vice-versa. Equivalent to the `!` operator.

- The argument can be a literal Boolean, a function returning a Boolean, or a reference to a column containing Boolean values.

Since the function returns a Boolean value, it can be used as a function or a conditional.

> **NOTE:** Within an expression, you might choose to use the corresponding operator, instead of this function. For more information, see *Logical Operators*.

**Wrangle vs. SQL:** This function is part of Wrangle , a proprietary data transformation language. Wrangle is not SQL. For more information, see *Wrangle Language*.

## Basic Usage

```
not(customerHappiness &gt; 70)
```

**Output:** If the value in the `customerHappiness` column is not greater than `70`, then the value is `true`. Otherwis e, the value is `false`.

## Syntax and Arguments

```
not(value1)
```

| Argument | Required? | Data Type | Description |
|----------|-----------|-----------|-------------|
| value1 | Y | string | The value must be a Boolean literal, column reference, or expression that evaluates to `true` or `false`. |

For more information on syntax standards, see *Language Documentation Syntax Notes*.

**value1**

Expression, column reference or literal that evaluates to a Boolean value.

- Missing or mismatched values generate missing results.

**Usage Notes:**

| Required? | Data Type | Example Value |
|-----------|-----------|---------------|
| Yes | Function or column reference returning a Boolean value or Boolean literal | `myHeight > 2.00` |

## Examples

> **Tip:** For additional examples, see *Common Tasks.*

### Example - Logical Functions

This example demonstrate the `AND`, `OR`, and `NOT` logical functions.

In this example, the dataset contains results from survey data on two questions about customers. The yes/no answers to each question determine if the customer is 1) still active, and 2) interested in a new offering.

**Functions:**

| Item | Description |
|---|---|
| AND Function | Returns `true` if both arguments evaluate to `true`. Equivalent to the `&&` operator. |
| OR Function | Returns `true` if either argument evaluates to `true`. Equivalent to the `││` operator. |
| NOT Function | Returns `true` if the argument evaluates to `false`, and vice-versa. Equivalent to the `!` operator. |

**Source:**

| Customer | isActive | isInterested |
|---|---|---|
| CustA | Y | Y |
| CustB | Y | N |
| CustC | N | Y |
| CustD | N | N |

**Transformation:**

Customers that are both active and interested should receive a phone call:

| | |
|---|---|
| **Transformation Name** | `New formula` |
| **Parameter: Formula type** | `Single row formula` |
| **Parameter: Formula** | `AND(isActive, isInterested)` |
| **Parameter: New column name** | `'phoneCall'` |

Customers that are either active or interested should receive an email:

| | |
|---|---|
| **Transformation Name** | `New formula` |
| **Parameter: Formula type** | `Single row formula` |
| **Parameter: Formula** | `OR(isActive, isInterested)` |
| **Parameter: New column name** | `'sendEmail'` |

Customers that are neither active or interested should be dropped from consideration for the offering:

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | AND(NOT(isActive),NOT(isInterested) |
| Parameter: New column name | 'dropCust' |

A savvy marketer might decide that if a customer receives a phone call, that customer should not be bothered with an email, as well:

| Transformation Name | Edit column with formula |
|---|---|
| Parameter: Columns | sendEmail |
| Parameter: Formula | IF(phoneCall == "TRUE", FALSE, sendEmail) |

**Results:**

| Customer | isActive | isInterested | dropCust | sendEmail | phoneCall |
|---|---|---|---|---|---|
| CustA | Y | Y | FALSE | FALSE | TRUE |
| CustB | Y | N | FALSE | TRUE | FALSE |
| CustC | N | Y | FALSE | TRUE | FALSE |
| CustD | N | N | TRUE | FALSE | FALSE |

# Comparison Functions

This category encompasses standard comparison operations, such as greater than and less than, and a set of functions to evaluate the presence of values within nested data.

# Comparison Operators

**Contents:**

Comparison operators enable you to compare values in the left-hand side of an expression to the values in the right-hand side of an expression.

```
(left-hand side) (operator) (right-hand side)
```

These evaluations result in a Boolean `true` or `false` result and can be used as the basis for determining whether the transformation is executed on the row or column of data. The following operators are supported:

| Operator Name | Symbol | Example Expression | Output | Notes |
|---|---|---|---|---|
| less than | < | `3 < 6` | `true` | |
| less than or equal to | <= | `6 <= 5` | `false` | The following operator generates an error: =< |
| greater than | > | `3 > 6` | `false` | |
| greater than or equal to | >= | `6 >= 5` | `true` | The following operator generates an error: => |
| equal to | == | `4 == 4` | `true` | For this comparison operator, you must use two equals signs, or an error is generated. |
| not equal to | <> or != | 4 <> 4 | `false` | Both operators are supported. The following operator generates an error: = ! |

The above examples apply to integer values only. Below, you can review how the comparison operators apply to different data types.

## Usage

Comparison operators are used to determine the condition of a set of data. Typically, they are applied in evaluations of values or rows.

For example, your dataset is the following:

| city |
|---|
| San Francisco |

| Los Angeles |
| Chicago |
| New York |

You could use the following transformation to flag all rows whose `city` value equals `San Francisco`:

| Transformation Name | `New formula` |
|---|---|
| **Parameter: Formula type** | `Single row formula` |
| **Parameter: Formula** | `(city == 'San Francisco')` |

Your output looks like the following:

| city | column1 |
|---|---|
| San Francisco | true |
| Los Angeles | false |
| Chicago | false |
| New York | false |

You can optionally combine the above with an `IF` function, which enables you to write values for `true` or `false` outcomes:

| Transformation Name | `New formula` |
|---|---|
| **Parameter: Formula type** | `Single row formula` |
| **Parameter: Formula** | `if(city == 'San Francisco', 'Home of the Giants!', 'Some other team')` |
| **Parameter: New column name** | `'BaseballTeam'` |

Note that the optional `as:` clause can be used to rename the generated columns. See *Derive Transform*.

| city | BaseballTeam |
|---|---|
| San Francisco | Home of the Giants! |
| Los Angeles | Some other team |
| Chicago | Some other team |
| New York | Some other team |

## Examples

> **Tip:** For additional examples, see *Common Tasks*.

> **NOTE:** When a comparison is applied to a set of values, the type of data of each source value is re-inferred to match any literal values used on the other side of the expression. This method allows for more powerful comparisons.
>
> In the following examples, values taken from the `MySource` column are re-typed to match the inferred data type of the other side of the comparison.

## Less Than (or Equal To)

| Column Type | Example Transformation | | Output | Notes |
|---|---|---|---|---|
| Integer | **Transformation Name** | `New formula` | • `true` for all values in `MySource` that are less than 5.<br>• Otherwise, `false`. | |
| | **Parameter: Formula type** | `Single row formula` | | |
| | **Parameter: Formula** | `(MySource < 5)` | | |

| Decimal | | | | Retains all rows in the dataset where the value in the `MySource` column is less than or equal to 2.5. | |
|---|---|---|---|---|---|
| | | **Transformation Name** | `Filter rows` | | |
| | | **Parameter: Condition** | `Custom formula` | | |
| | | **Parameter: Type of formula** | `Custom single` | | |
| | | **Parameter: Condition** | `(MySource <= 2.5)` | | |
| | | **Parameter: Action** | `Keep matching rows` | | |

| Datetime | | | | Retains all rows whose `Date` column value is less than or equal to `12/31/2009`. | You can also use the `DATEDIF` function to generate the number of days difference between two date values. Then, you can compare this difference to another value. See *DATEDIF Function*. |
|----------|--|--|--|--|--|
| | **Transformation Name** | `Filter rows` | | | |
| | **Parameter: Condition** | `Custom formula` | | | |
| | **Parameter: Type of formula** | `Custom single` | | | |
| | **Parameter: Condition** | `(Date <= DATE(2009,12,31))` | | | |
| | **Parameter: Action** | `Keep matching rows` | | | |

| Column Type | Example Transformation | | Output | Notes |
|---|---|---|---|---|
| String (and all other data types) | **Transformation Name** | New formula | • true for any string value in the MySource column whose length is less than 5 characters.<br>• Otherwise, false<br>• See *LEN Function*. | • For comparison purposes, all data types not previously listed in this table behave like strings.<br>• Since strings are non-numeric value, a function must be applied to string data to render a comparison. |
| | **Parameter: Formula type** | Single row formula | | |
| | **Parameter: Formula** | (LEN(MySource) < 5)) | | |

## Greater Than (or Equal To)

See previous section.

## Equal to

| Column Type | Example Transformation | | Output | Notes |
|---|---|---|---|---|
| Integer | **Transformation Name** | New formula | • true for all values in the MySource column that are 5.<br>• Otherwise, false. | If the source column contains Decimal values and the right-hand side is an integer value, the Decimal values that are also integers can match in the comparison (e.g. 2.0 == 2). |
| | **Parameter: Formula type** | Single row formula | | |
| | **Parameter: Formula** | (MySource == 5) | | |

| Decimal | | | | Retains all rows in the dataset where the value in the `MySou rce` column is exactly 2.5. | If the source column contains integers and the right-hand side is a Decimal type value, integer values are rounded for comparison. |
|---------|---|---|---|---|---|
| | | **Transformation Name** | `Fi lt er ro ws` | | |
| | | **Parameter: Condition** | `Cu st om fo rm ula` | | |
| | | **Parameter: Type of formula** | `Cu st om si ng le` | | |
| | | **Parameter: Condition** | `(M yS ou rc e == 2. 5)` | | |
| | | **Parameter: Action** | `Ke ep ma tc hi ng ro ws` | | |

| Datetime | | | | Retains all rows whose `Date` column value is equal to `12/25/2016`. | |
|---|---|---|---|---|---|
| | | **Transformation Name** | `Filter rows` | | |
| | | **Parameter: Condition** | `Custom formula` | | |
| | | **Parameter: Type of formula** | `Custom single` | | |
| | | **Parameter: Condition** | `(Date == DATE(2016,12,25))` | | |
| | | **Parameter: Action** | `Keep matching rows` | | |

| Column Type | | | Example Transformation | Output | Notes |
|---|---|---|---|---|---|
| String (and all other data types) | | | **Transformation Name** | `Filter rows` | Retains all rows in the dataset where the length of the string value in the `MySource` column is 5 characters. | • For comparison purposes, all data types not previously listed in this table behave like strings.<br>• Since strings are non-numeric value, a function must be applied to string data to render a comparison. |
| | | | **Parameter: Condition** | `Custom formula` | | |
| | | | **Parameter: Type of formula** | `Custom single` | | |
| | | | **Parameter: Condition** | `(LEN(MySource)==5))` | | |
| | | | **Parameter: Action** | `Keep matching rows` | | |

## Not Equal to

| Column Type | Example Transformation | Output | Notes |
|---|---|---|---|

| Integer | | | | If the source column contains Decimal values and the right-hand side is an integer value, the Decimal values that are also integers can match in the comparison (e.g. $2.0 == 2$). |
|---|---|---|---|---|
| | **Transformation Name** | `New formula` | • `true` for all values in the `MySource` column that are not 5.<br>• Otherwise, `false`. | |
| | **Parameter: Formula type** | `Single row formula` | | |
| | **Parameter: Formula** | `(MySource <> 5)` | | |

| Decimal | | | | Retains all rows in the dataset where the value in the `MySource` column is not 2.5. | If the source column contains integers and the right-hand side is a Decimal type value, integer values are rounded for comparison. |
|---|---|---|---|---|---|
| | | **Transformation Name** | `Filter rows` | | |
| | | **Parameter: Condition** | `Custom formula` | | |
| | | **Parameter: Type of formula** | `Custom single` | | |
| | | **Parameter: Condition** | `(MySource <> 2.5)` | | |
| | | **Parameter: Action** | `Keep matching rows` | | |

| Datetime | | | | Retains all rows in the dataset where the `Date` value does not equal 4/15/2016. | |
|---|---|---|---|---|---|
| | | **Transformation Name** | `Filter rows` | | |
| | | **Parameter: Condition** | `Custom formula` | | |
| | | **Parameter: Type of formula** | `Custom single` | | |
| | | **Parameter: Condition** | `(Date <> DATE(2016, 4, 15))` | | |
| | | **Parameter: Action** | `Keep matching rows` | | |

| | | | Retains all rows in the dataset where the length of the string value in the `MySource` column is not 5 characters. | • For comparison purposes, all data types not previously listed in this table behave like strings.<br>• Since strings are non-numeric value, a function must be applied to string data to render a comparison. |
|---|---|---|---|---|
| String (and all other data types) | **Transformation Name** | `Filter rows` | | |
| | **Parameter: Condition** | `Custom formula` | | |
| | **Parameter: Type of formula** | `Custom single` | | |
| | **Parameter: Condition** | `(LEN(MySource)<>5))` | | |
| | **Parameter: Action** | `Keep matching rows` | | |

# ISEVEN Function

Returns `true` if the argument is an even value. Argument can be an Integer, a function returning Integers, or a column reference.

Since the function returns a Boolean value, it can be used as a function or a conditional.

**Wrangle vs. SQL:** This function is part of Wrangle , a proprietary data transformation language. Wrangle is not SQL. For more information, see *Wrangle Language*.

## Basic Usage

**Integer literal value:**

```
iseven(&apos;4&apos;)
```

**Output:** Returns the value `true`.

**Column reference value:**

```
iseven(errorCount)
```

**Output:** Returns `true` if the value in the `errorCount` column is an even number.

## Syntax and Arguments

```
iseven(int_value)
```

| Argument | Required? | Data Type | Description |
|----------|-----------|-----------|-------------|
| int_value | Y | integer | This value can be an Integer, a function returning an Integer, or a column reference. |

For more information on syntax standards, see *Language Documentation Syntax Notes*.

**int_value**

Name of the columns, expressions, or literals to compare.

- Missing values generate missing string results.

**Usage Notes:**

| Required? | Data Type | Example Value |
|-----------|-----------|---------------|
| Yes | Column reference, function, or Integer literal value | `myColumn` |

## Examples

> **Tip:** For additional examples, see *Common Tasks*.

**Example - Basic Equal and Notequal Functions**

This example demonstrates comparison functions.

**Functions:**

| Item | Description |
|---|---|
| EQUAL Function | Returns `true` if the first argument is equal to the second argument. Equivalent to the = operator. |
| NOTEQUAL Function | Returns `true` if the first argument is not equal to the second argument. Equivalent to the `<>` or `!=` operator. |
| ISEVEN Function | Returns `true` if the argument is an even value. Argument can be an Integer, a function returning Integers, or a column reference. |
| ISODD Function | Returns `true` if the argument is an odd value. Argument can be an Integer, a function returning Integers, or a column reference. |

In this example, the dataset contains current measurements of the sides of rectangular areas next to the size of those areas as previously reported. Using these functions, you can perform some light analysis of the data.

**Source:**

| sideA | sideB | reportedArea |
|---|---|---|
| 4 | 14 | 56 |
| 6 | 6 | 35 |
| 8 | 4 | 32 |
| 15 | 15 | 200 |
| 4 | 7 | 28 |
| 12 | 6 | 70 |
| 9 | 9 | 81 |

**Transformation:**

In the first test, you are determining if the four-sided area is a square, based on a comparison of the measured values for `sideA` and `sideB`:

| Transformation Name | New formula |
|---|---|
| **Parameter: Formula type** | Single row formula |
| **Parameter: Formula** | EQUAL(sideA, sideB) |
| **Parameter: New column name** | 'isSquare' |

Next, you can use the reported sides to calculate the area of the shape and compare it to the area previously reported:

| Transformation Name | New formula |
|---|---|
| **Parameter: Formula type** | Single row formula |
| **Parameter: Formula** | NOTEQUAL(sideA * sideB, reportedArea) |

| Parameter: New column name | 'isValidData' |
|---|---|

You can also compute if the reportedArea can be divided into even square units:

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | ISEVEN(reportedArea) |
| Parameter: New column name | 'isReportedAreaEven' |

You can test if either measured side is an odd number of units:

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | IF((ISODD(sideA) == true) OR (ISODD(sideB) == true),TRUE, FALSE) |
| Parameter: New column name | 'isSideOdd' |

**Results:**

| sideA | sideB | reportedArea | isSquare | isValidData | isReportedAreaEven | isSideOdd |
|---|---|---|---|---|---|---|
| 4 | 14 | 56 | FALSE | FALSE | TRUE | FALSE |
| 6 | 6 | 35 | TRUE | TRUE | TRUE | FALSE |
| 8 | 4 | 32 | FALSE | FALSE | TRUE | FALSE |
| 15 | 15 | 200 | TRUE | TRUE | TRUE | TRUE |
| 4 | 7 | 28 | FALSE | FALSE | TRUE | TRUE |
| 12 | 6 | 70 | FALSE | TRUE | TRUE | FALSE |
| 9 | 9 | 81 | TRUE | FALSE | FALSE | FALSE |

# ISODD Function

Returns `true` if the argument is an odd value. Argument can be an Integer, a function returning Integers, or a column reference.

Since the function returns a Boolean value, it can be used as a function or a conditional.

**Wrangle vs. SQL:** This function is part of Wrangle , a proprietary data transformation language. Wrangle is not SQL. For more information, see *Wrangle Language*.

## Basic Usage

**Integer literal value:**

```
isodd(&apos;3&apos;)
```

**Output:** Returns the value `true`.

**Column reference value:**

```
isodd(countStudents)
```

**Output:** If the value in the `countStudents` column is an odd number, then return `true`.

## Syntax and Arguments

```
isodd(int_value)
```

| Argument | Required? | Data Type | Description |
|----------|-----------|-----------|-------------|
| int_value | Y | integer | This value can be an Integer, a function returning an Integer, or a column reference. |

For more information on syntax standards, see *Language Documentation Syntax Notes*.

**int_value**

Name of the columns, expressions, or literals to compare.

- Missing values generate missing string results.

**Usage Notes:**

| Required? | Data Type | Example Value |
|-----------|-----------|---------------|
| Yes | Column reference, function, or Integer literal value | `myColumn` |

## Examples

> **Tip:** For additional examples, see *Common Tasks*.

### Example - Basic Equal and Notequal Functions

This example demonstrates comparison functions.

**Functions:**

| Item | Description |
|---|---|
| EQUAL Function | Returns `true` if the first argument is equal to the second argument. Equivalent to the = operator. |
| NOTEQUAL Function | Returns `true` if the first argument is not equal to the second argument. Equivalent to the `<>` or `!=` operator. |
| ISEVEN Function | Returns `true` if the argument is an even value. Argument can be an Integer, a function returning Integers, or a column reference. |
| ISODD Function | Returns `true` if the argument is an odd value. Argument can be an Integer, a function returning Integers, or a column reference. |

In this example, the dataset contains current measurements of the sides of rectangular areas next to the size of those areas as previously reported. Using these functions, you can perform some light analysis of the data.

**Source:**

| sideA | sideB | reportedArea |
|---|---|---|
| 4 | 14 | 56 |
| 6 | 6 | 35 |
| 8 | 4 | 32 |
| 15 | 15 | 200 |
| 4 | 7 | 28 |
| 12 | 6 | 70 |
| 9 | 9 | 81 |

**Transformation:**

In the first test, you are determining if the four-sided area is a square, based on a comparison of the measured values for `sideA` and `sideB`:

| Transformation Name | New formula |
|---|---|
| **Parameter: Formula type** | Single row formula |
| **Parameter: Formula** | EQUAL(sideA, sideB) |
| **Parameter: New column name** | 'isSquare' |

Next, you can use the reported sides to calculate the area of the shape and compare it to the area previously reported:

| Transformation Name | New formula |
|---|---|
| **Parameter: Formula type** | Single row formula |
| **Parameter: Formula** | NOTEQUAL(sideA * sideB, reportedArea) |

| Parameter: New column name | 'isValidData' |
|---|---|

You can also compute if the reportedArea can be divided into even square units:

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | ISEVEN(reportedArea) |
| Parameter: New column name | 'isReportedAreaEven' |

You can test if either measured side is an odd number of units:

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | IF((ISODD(sideA) == true) OR (ISODD(sideB) == true),TRUE, FALSE) |
| Parameter: New column name | 'isSideOdd' |

**Results:**

| sideA | sideB | reportedArea | isSquare | isValidData | isReportedAreaEven | isSideOdd |
|---|---|---|---|---|---|---|
| 4 | 14 | 56 | FALSE | FALSE | TRUE | FALSE |
| 6 | 6 | 35 | TRUE | TRUE | TRUE | FALSE |
| 8 | 4 | 32 | FALSE | FALSE | TRUE | FALSE |
| 15 | 15 | 200 | TRUE | TRUE | TRUE | TRUE |
| 4 | 7 | 28 | FALSE | FALSE | TRUE | TRUE |
| 12 | 6 | 70 | FALSE | TRUE | TRUE | FALSE |
| 9 | 9 | 81 | TRUE | FALSE | FALSE | FALSE |

# IN Function

**Contents:**

---

Returns `true` if the first parameter is contained in the array of values in the second parameter.

- The value to match can be a literal or a reference to a column.
- The second parameter must be in array format.

Since the `IN` function returns a Boolean value, it can be used as a function or a conditional.

> **Tip:** When you select values in a histogram for a column of String type, the function that identifies the values on which to perform a transform is typically `IN`.

> **Tip:** If you need the location of the matched value within the source, use the `FIND` function. See *FIND Function.*

**Wrangle vs. SQL:** This function is part of Wrangle , a proprietary data transformation language. Wrangle is not SQL. For more information, see *Wrangle Language*.

## Basic Usage

```
in(brand, ['discount','mid','high-end'])
```

**Output:** Returns `true` if the value in the `brand` column is either `discount`, `mid`, or `high-end`.

## Syntax and Arguments

```
in(column_string, values_array)
```

| Argument | Required? | Data Type | Description |
|----------|-----------|-----------|-------------|
| column_string | Y | string | Name of column or literal to locate in the column specified in the second parameter |
| values_array | Y | array literal | Array literal of values to search |

For more information on syntax standards, see *Language Documentation Syntax Notes*.

**column_string**

Name of the column or literal to find in the second parameter.

- Missing values generate missing string results.

- String constants must be quoted (`'Hello, World'`).
- Multiple columns and wildcards are not supported.

**Usage Notes:**

| Required? | Data Type | Example Value |
|---|---|---|
| Yes | Column reference or any value | `myColumn` |

**values_array**

Array of values to search for the first parameter.

- Column references are not supported.

**Usage Notes:**

| Required? | Data Type | Example Value |
|---|---|---|
| Yes | Array literal | `'Steve'` |

## Examples

> **Tip:** For additional examples, see *Common Tasks*.

**Example - Replace T-shirt color**

**Source:**

You have the following source data on your products:

| ProdId | ProductName | Color | Brand |
|---|---|---|---|
| P001 | T-shirt | white | discount |
| P002 | pants | beige | discount |
| P003 | hat | black | discount |
| P004 | T-shirt | white | mid |
| P005 | pants | black | mid |
| P006 | hat | red | mid |
| P007 | T-shirt | white | high-end |
| P008 | pants | white | high-end |
| P009 | hat | blue | high-end |

In the data, you notice an error. For the discount and mid brands, T-shirt color should be `orange`. You need to fix it.

**Transformation:**

In the Transformer page, you select the `white` value from the histogram at the top of the `Color` column. Among the suggestion cards, select the Set transform. For the first variant, all values are missing. Click **Modify**. The current transform is the following:

| Transformation Name | Edit column with formula |
|---|---|
| Parameter: Columns | Color |
| Parameter: Formula | null() |
| Parameter: Group rows by | Color == 'white' |

In the Preview, you can see that this transform matches all `white` values in the column and replaces them with a null value. Since the replacement value is `orange`, you can edit the transform so it looks like the following:

| Transformation Name | Edit column with formula |
|---|---|
| Parameter: Columns | Color |
| Parameter: Formula | 'orange' |
| Parameter: Group rows by | Color == 'white' |

This step looks better. However, it is replacing all instances of `white`, including those for white pants (P008) and high-end T-shirts (p007), which should not be replaced. To fix, you must add conditions to the `row` expression. First, add the following, which ensures that the transform only replaces for T-shirts:

| Transformation Name | Edit column with formula |
|---|---|
| Parameter: Columns | Color |
| Parameter: Formula | 'orange' |
| Parameter: Group rows by | (Color == 'white' && ProductName == 'T-shirt') |

Now, the Preview shows that only T-shirt values are being changed. The transform needs to be further modified to restrict only to the appropriate brands (`discount` and `mid`):

| Transformation Name | Edit column with formula |
|---|---|
| Parameter: Columns | Color |
| Parameter: Formula | 'orange' |
| Parameter: Group rows by | (Color == 'white' && ProductName == 'T-shirt' && IN(Brand, ["discount","mid"])) |

> **NOTE:** It's possible to specify the brand restriction as `(Brand <> 'high-end')`. However, if there are other brand values in the full dataset, this restriction fails.

**Results:**

| ProdId | ProductName | Color | Brand |
|---|---|---|---|

| | | | |
|---|---|---|---|
| P001 | T-shirt | orange | discount |
| P002 | pants | beige | discount |
| P003 | hat | black | discount |
| P004 | T-shirt | orange | mid |
| P005 | pants | black | mid |
| P006 | hat | red | mid |
| P007 | T-shirt | white | high-end |
| P008 | pants | white | high-end |
| P009 | hat | blue | high-end |

# MATCHES Function

**Contents:**

---

Returns `true` if a value contains a string or pattern. The value to search can be a string literal, a function returning a string, or a reference to a column of String type.

Since the `MATCHES` function returns a Boolean value, it can be used as both a function and as a conditional.

> **Tip:** When you select values in a histogram for a column of Array type, the function that identifies the values on which to perform a transform is typically `MATCHES`.

> **Tip:** If you need the location of the matched string within the source, use the `FIND` function. See *FIND Function.*

**Wrangle vs. SQL:** This function is part of Wrangle , a proprietary data transformation language. Wrangle is not SQL. For more information, see *Wrangle Language.*

## Basic Usage

**Column reference example:**

```
matches(ProdId, &apos;Fun Toy&apos;)
```

**Output:** Returns `true` when the value in the `ProdId` column value contains the string literal `Fun Toy`.

**String literal example:**

```
matches(&apos;Hello, World&apos;, &apos;Hello&apos;)
```

**Output:** Returns `true`.

## Syntax and Arguments

```
matches(column_string,string_pattern <span>[,ignore_case]</span>)
```

| Argument | Required? | Data Type | Description |
|---|---|---|---|
| column_string | Y | string | Name of column or string literal to be searched |
| string_pattern | Y | string | Name of column, function returning a string ,or string literal or pattern to find |

| ignore_case | N | string | When `true`, matching is case-insensitive. Default is `false`. |
|---|---|---|---|

For more information on syntax standards, see *Language Documentation Syntax Notes*.

### column_string

Name of the column or string literal to be searched.

- Missing string or column values generate missing string results.
    - String constants must be quoted (`'Hello, World'`).
- Multiple columns can be specified as an array ( `matches([Col1,Col2],'hello')`.

#### Usage Notes:

| Required? | Data Type | Example Value |
|---|---|---|
| Yes | String | `MyColumn` |

### string_pattern

Column of strings, function returning a string, or string literal. Value can be a column reference, string literal, Patt ern , or regular expression to match against the source column-string.

#### Usage Notes:

| Required? | Data Type | Example Value |
|---|---|---|
| Yes | Column reference or string literal or pattern | `'home page'` |

### ignore_case

When `true`, matches are case-insensitive. Default is `false`.

> **NOTE:** This argument is not required. By default, matches are case-sensitive.

| Required? | Data Type | Example Value |
|---|---|---|
| No | String literal evaluating to a Boolean | `'true'` |

## Examples

> **Tip:** For additional examples, see *Common Tasks*.

### Example - Filtering log data

In downloaded log files, you might see error messages of the following type:

- `INFO` - status information on the process
- `WARNING` - system encountered a non-fatal error during execution
- `ERROR` - system encountered an error, which might have caused the job to fail.

For purposes of analysis, you might want to filter out the data for `INFO` and `WARNING` messages.

**Source:**

Here is example data from a log file of a failed job:

| log |
| --- |
| 2016-01-29T00:14:24.924Z com.example.hadoopdata.monitor.spark_runner.ProfilerServiceClient [pool-13-thread-1] INFO com.example. hadoopdata.monitor.spark_runner.BatchProfileSparkRunner - Spark Profiler URL - http://localhost:4006/ |
| 2016-01-29T00:14:40.066Z com.example.hadoopdata.monitor.spark_runner.BatchProfileSparkRunner [pool-13-thread-1] INFO com. example.hadoopdata.monitor.spark_runner.BatchProfileSparkRunner - Spark process ID was null. |
| 2016-01-29T00:14:40.067Z com.example.hadoopdata.monitor.spark_runner.BatchProfileSparkRunner [pool-13-thread-1] INFO com. example.hadoopdata.monitor.spark_runner.BatchProfileSparkRunner - -------------------------------END SPARK JOB----------------------------- |
| 2016-01-29T00:14:44.961Z com.example.hadoopdata.joblaunch.server.BatchPollingWorker [pool-4-thread-2] ERROR com.example. hadoopdata.joblaunch.server.BatchPollingWorker - Job '128' threw an exception during execution |
| 2016-01-29T00:14:44.962Z com.example.hadoopdata.joblaunch.server.BatchPollingWorker [pool-4-thread-2] INFO com.example. hadoopdata.joblaunch.server.BatchPollingWorker - Making sure async worker is stopped |
| 2016-01-29T00:14:44.962Z com.example.hadoopdata.joblaunch.server.BatchPollingWorker [pool-4-thread-2] INFO com.example. hadoopdata.joblaunch.server.BatchPollingWorker - Notifying monitor for job '128', code 'FAILURE' |
| 2016-01-29T00:14:44.988Z com.example.hadoopdata.monitor.client.MonitorClient [pool-4-thread-2] INFO com.example.hadoopdata. monitor.client.MonitorClient - Request succeeded to monitor ip-0-0-0-0.example.com:8001 |

**Transformation:**

When the above data is loaded into the application, you might want to break up the data into separate columns, which splits them on the `Z` character at the end of the timestamp:

| | |
| --- | --- |
| **Transformation Name** | `Split column` |
| **Parameter: Column** | `column1` |
| **Parameter: Option** | `On pattern` |
| **Parameter: Match pattern** | `` `Z ` `` |

Then, you can rename the two columns: `Timestamp` and `Log_Message`. To filter out the `INFO` and `WARNING` messages, you can use the following transforms, which match on the string literals to identify these messages:

| | |
| --- | --- |
| **Transformation Name** | `Filter rows` |
| **Parameter: Condition** | `Custom formula` |
| **Parameter: Type of formula** | `Custom single` |
| **Parameter: Condition** | `matches(Log_Message, '] INFO ')` |
| **Parameter: Action** | `Delete matching rows` |

| | |
| --- | --- |
| **Transformation Name** | `Filter rows` |
| **Parameter: Condition** | `Custom formula` |
| **Parameter: Type of formula** | `Custom single` |

| Parameter: Condition | `matches(Log_Message, '] WARNING ')` |
|---|---|
| Parameter: Action | `Delete matching rows` |

**Results:**

After the above steps, the data should look like the following:

| Timestamp | Log_Message |
|---|---|
| 2016-01-29T00:14:44.961 | com.example.hadoopdata.joblaunch.server.BatchPollingWorker [pool-4-thread-2] ERROR com.example.hadoopdata.joblaunch.server.BatchPollingWorker - Job '128' threw an exception during execution |

# EQUAL Function

Returns `true` if the first argument is equal to the second argument. Equivalent to the `=` operator.

- Each argument can be a literal String, Integer or Decimal number, a function, or a column reference.

Since the function returns a Boolean value, it can be used as a function or a conditional.

> **NOTE:** Within an expression, you might choose to use the corresponding operator, instead of this function. For more information, see *Comparison Operators*.

**Wrangle vs. SQL:** This function is part of Wrangle , a proprietary data transformation language. Wrangle is not SQL. For more information, see *Wrangle Language*.

## Basic Usage

```
if(equal(errorCount, 0),&apos;ok&apos;,&apos;Error_recorded&apos;)
```

**Output:** If the value in the `errorCount` column is zero, then the `status` column value is `ok`. Otherwise, the value is `Error_recorded`.

## Syntax and Arguments

```
equal(value1, value2)
```

| Argument | Required? | Data Type | Description |
|----------|-----------|-----------|-------------|
| value1 | Y | string | The first value. This value can be a String, a number, a function, or a column reference. |
| value2 | Y | string | The second value. This value can be a String, a number, a function, or a column reference. |

For more information on syntax standards, see *Language Documentation Syntax Notes*.

**value1, value2**

Names of the columns, expressions, or literals to compare.

- Missing values generate missing string results.

**Usage Notes:**

| Required? | Data Type | Example Value |
|-----------|-----------|---------------|
| Yes | Column reference, function, or numeric or String value | `myColumn` |

## Examples

> **Tip:** For additional examples, see *Common Tasks*.

### Example - Basic Equal and Notequal Functions

This example demonstrates comparison functions.

**Functions:**

| Item | Description |
|---|---|
| EQUAL Function | Returns `true` if the first argument is equal to the second argument. Equivalent to the = operator. |
| NOTEQUAL Function | Returns `true` if the first argument is not equal to the second argument. Equivalent to the <> or != operator. |
| ISEVEN Function | Returns `true` if the argument is an even value. Argument can be an Integer, a function returning Integers, or a column reference. |
| ISODD Function | Returns `true` if the argument is an odd value. Argument can be an Integer, a function returning Integers, or a column reference. |

In this example, the dataset contains current measurements of the sides of rectangular areas next to the size of those areas as previously reported. Using these functions, you can perform some light analysis of the data.

**Source:**

| sideA | sideB | reportedArea |
|---|---|---|
| 4 | 14 | 56 |
| 6 | 6 | 35 |
| 8 | 4 | 32 |
| 15 | 15 | 200 |
| 4 | 7 | 28 |
| 12 | 6 | 70 |
| 9 | 9 | 81 |

**Transformation:**

In the first test, you are determining if the four-sided area is a square, based on a comparison of the measured values for `sideA` and `sideB`:

| Transformation Name | `New formula` |
|---|---|
| **Parameter: Formula type** | `Single row formula` |
| **Parameter: Formula** | `EQUAL(sideA, sideB)` |
| **Parameter: New column name** | `'isSquare'` |

Next, you can use the reported sides to calculate the area of the shape and compare it to the area previously reported:

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | NOTEQUAL(sideA * sideB, reportedArea) |
| Parameter: New column name | 'isValidData' |

You can also compute if the reportedArea can be divided into even square units:

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | ISEVEN(reportedArea) |
| Parameter: New column name | 'isReportedAreaEven' |

You can test if either measured side is an odd number of units:

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | IF((ISODD(sideA) == true) OR (ISODD(sideB) == true),TRUE, FALSE) |
| Parameter: New column name | 'isSideOdd' |

**Results:**

| sideA | sideB | reportedArea | isSquare | isValidData | isReportedAreaEven | isSideOdd |
|---|---|---|---|---|---|---|
| 4 | 14 | 56 | FALSE | FALSE | TRUE | FALSE |
| 6 | 6 | 35 | TRUE | TRUE | TRUE | FALSE |
| 8 | 4 | 32 | FALSE | FALSE | TRUE | FALSE |
| 15 | 15 | 200 | TRUE | TRUE | TRUE | TRUE |
| 4 | 7 | 28 | FALSE | FALSE | TRUE | TRUE |
| 12 | 6 | 70 | FALSE | TRUE | TRUE | FALSE |
| 9 | 9 | 81 | TRUE | FALSE | FALSE | FALSE |

# NOTEQUAL Function

Returns `true` if the first argument is not equal to the second argument. Equivalent to the `<>` or `!=` operator.

- Each argument can be a literal String, Integer or Decimal number, a function, or a column reference.

Since the function returns a Boolean value, it can be used as a function or a conditional.

> **NOTE:** Within an expression, you might choose to use the corresponding operator, instead of this function. For more information, see *Comparison Operators*.

**Wrangle vs. SQL:** This function is part of Wrangle , a proprietary data transformation language. Wrangle is not SQL. For more information, see *Wrangle Language*.

## Basic Usage

```
notequal(myValue, checkSum)
```

**Output:** Returns `true` if the value in the `myValue` column does not equal the value in the `checkSum` column.

## Syntax and Arguments

```
notequal(value1, value2)
```

| Argument | Required? | Data Type | Description |
|----------|-----------|-----------|-------------|
| value1 | Y | string | The first value. This value can be a String, a number, a function, or a column reference. |
| value2 | Y | string | The second value. This value can be a String, a number, a function, or a column reference. |

For more information on syntax standards, see *Language Documentation Syntax Notes*.

**value1, value2**

Names of the columns, expressions, or literals to compare.

- Missing values generate missing results.

**Usage Notes:**

| Required? | Data Type | Example Value |
|-----------|-----------|---------------|
| Yes | Column reference, function, or numeric or String value | `myColumn` |

## Examples

> **Tip:** For additional examples, see *Common Tasks*.

### Example - Basic Equal and Notequal Functions

This example demonstrates comparison functions.

**Functions:**

| Item | Description |
|------|-------------|
| EQUAL Function | Returns `true` if the first argument is equal to the second argument. Equivalent to the = operator. |
| NOTEQUAL Function | Returns `true` if the first argument is not equal to the second argument. Equivalent to the <> or != operator. |
| ISEVEN Function | Returns `true` if the argument is an even value. Argument can be an Integer, a function returning Integers, or a column reference. |
| ISODD Function | Returns `true` if the argument is an odd value. Argument can be an Integer, a function returning Integers, or a column reference. |

In this example, the dataset contains current measurements of the sides of rectangular areas next to the size of those areas as previously reported. Using these functions, you can perform some light analysis of the data.

**Source:**

| sideA | sideB | reportedArea |
|-------|-------|--------------|
| 4 | 14 | 56 |
| 6 | 6 | 35 |
| 8 | 4 | 32 |
| 15 | 15 | 200 |
| 4 | 7 | 28 |
| 12 | 6 | 70 |
| 9 | 9 | 81 |

**Transformation:**

In the first test, you are determining if the four-sided area is a square, based on a comparison of the measured values for `sideA` and `sideB`:

| Transformation Name | `New formula` |
|---------------------|---------------|
| Parameter: Formula type | `Single row formula` |
| Parameter: Formula | `EQUAL(sideA, sideB)` |
| Parameter: New column name | `'isSquare'` |

Next, you can use the reported sides to calculate the area of the shape and compare it to the area previously reported:

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | NOTEQUAL(sideA * sideB, reportedArea) |
| Parameter: New column name | 'isValidData' |

You can also compute if the reportedArea can be divided into even square units:

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | ISEVEN(reportedArea) |
| Parameter: New column name | 'isReportedAreaEven' |

You can test if either measured side is an odd number of units:

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | IF((ISODD(sideA) == true) OR (ISODD(sideB) == true),TRUE, FALSE) |
| Parameter: New column name | 'isSideOdd' |

**Results:**

| sideA | sideB | reportedArea | isSquare | isValidData | isReportedAreaEven | isSideOdd |
|---|---|---|---|---|---|---|
| 4 | 14 | 56 | FALSE | FALSE | TRUE | FALSE |
| 6 | 6 | 35 | TRUE | TRUE | TRUE | FALSE |
| 8 | 4 | 32 | FALSE | FALSE | TRUE | FALSE |
| 15 | 15 | 200 | TRUE | TRUE | TRUE | TRUE |
| 4 | 7 | 28 | FALSE | FALSE | TRUE | TRUE |
| 12 | 6 | 70 | FALSE | TRUE | TRUE | FALSE |
| 9 | 9 | 81 | TRUE | FALSE | FALSE | FALSE |

# GREATERTHAN Function

**Contents:**

---

Returns `true` if the first argument is greater than but not equal to the second argument. Equivalent to the `>` operator.

- Each argument can be a literal Integer or Decimal number, a function returning a number, or a reference to a column that contains numbers.

Since the function returns a Boolean value, it can be used as a function or a conditional.

> **NOTE:** Within an expression, you might choose to use the corresponding operator, instead of this function. For more information, see *Comparison Operators*.

**Wrangle vs. SQL:** This function is part of Wrangle , a proprietary data transformation language. Wrangle is not SQL. For more information, see *Wrangle Language*.

## Basic Usage

```
greaterthan(Errors, 10)
```

**Output:** Returns `true` when the value in the `Errors` column is greater than 10.

## Syntax and Arguments

```
greaterthan(value1, value2)
```

| Argument | Required? | Data Type | Description |
|----------|-----------|-----------|-------------|
| value1 | Y | string | The first value. This can be a number, a function returning a number, or a column containing numbers. |
| value2 | Y | string | The second value. This can be a number, a function returning a number, or a column containing numbers. |

For more information on syntax standards, see *Language Documentation Syntax Notes*.

### value1, value2

Names of the column, expressions, or literals to compare.

- Missing values generate missing string results.

**Usage Notes:**

| Required? | Data Type | Example Value |
|---|---|---|
| Yes | Column reference, function, or numeric or String value | `myColumn` |

## Examples

> **Tip:** For additional examples, see *Common Tasks*.

**Example - Basic Comparison Functions**

This example illustrates the comparison functions in Designer Cloud Powered by Trifacta® Enterprise Edition.

**Functions**:

| Item | Description |
|---|---|
| LESSTHAN Function | Returns `true` if the first argument is less than but not equal to the second argument. Equivalent to the < operator. |
| LESSTHANEQUAL Function | Returns `true` if the first argument is less than or equal to the second argument. Equivalent to the <= operator. |
| EQUAL Function | Returns `true` if the first argument is equal to the second argument. Equivalent to the = operator. |
| NOTEQUAL Function | Returns `true` if the first argument is not equal to the second argument. Equivalent to the <> or != operator. |
| GREATERTHAN Function | Returns `true` if the first argument is greater than but not equal to the second argument. Equivalent to the > operator. |
| GREATERTHANEQUAL Function | Returns `true` if the first argument is greater than or equal to the second argument. Equivalent to the >= operator. |

**Source:**

| colA | colB |
|---|---|
| 1 | 11 |
| 2 | 10 |
| 3 | 9 |
| 4 | 8 |
| 5 | 7 |
| 6 | 6 |
| 7 | 5 |
| 8 | 4 |
| 9 | 3 |

| 10 | 2 |
|----|---|
| 11 | 1 |

**Transformation:**

Add the following transforms to your recipe, one for each comparison function:

| Transformation Name | New formula |
|---|---|
| **Parameter: Formula type** | Single row formula |
| **Parameter: Formula** | LESSTHAN(colA, colB) |
| **Parameter: New column name** | 'lt' |

| Transformation Name | New formula |
|---|---|
| **Parameter: Formula type** | Single row formula |
| **Parameter: Formula** | LESSTHANEQUAL(colA, colB) |
| **Parameter: New column name** | 'lte' |

| Transformation Name | New formula |
|---|---|
| **Parameter: Formula type** | Single row formula |
| **Parameter: Formula** | EQUAL(colA, colB) |
| **Parameter: New column name** | 'eq' |

| Transformation Name | New formula |
|---|---|
| **Parameter: Formula type** | Single row formula |
| **Parameter: Formula** | NOTEQUAL(colA, colB) |
| **Parameter: New column name** | 'neq' |

| Transformation Name | New formula |
|---|---|
| **Parameter: Formula type** | Single row formula |
| **Parameter: Formula** | GREATERTHAN(colA, colB) |
| **Parameter: New column name** | 'gt' |

| Transformation Name | New formula |
|---|---|
| **Parameter: Formula type** | Single row formula |
| **Parameter: Formula** | GREATERTHANEQUAL(colA, colB) |

| Parameter: New column name | `'gte'` |
|---|---|

**Results:**

| colA | colB | gte | gt | neq | eq | lte | lt |
|---|---|---|---|---|---|---|---|
| 1 | 11 | false | false | true | false | true | true |
| 2 | 10 | false | false | true | false | true | true |
| 3 | 9 | false | false | true | false | true | true |
| 4 | 8 | false | false | true | false | true | true |
| 5 | 7 | false | false | true | false | true | true |
| 6 | 6 | true | false | false | true | true | false |
| 7 | 5 | true | true | true | false | false | false |
| 8 | 4 | true | true | true | false | false | false |
| 9 | 3 | true | true | true | false | false | false |
| 10 | 2 | true | true | true | false | false | false |
| 11 | 1 | true | true | true | false | false | false |

### Example - Using Comparisons to Test Ranges

This example demonstrates functions for comparing the relative values of two functions.

**Functions:**

| Item | Description |
|---|---|
| LESSTHANEQUAL Function | Returns `true` if the first argument is less than or equal to the second argument. Equivalent to the `<=` operator. |
| LESSTHAN Function | Returns `true` if the first argument is less than but not equal to the second argument. Equivalent to the `<` operator. |
| GREATERTHANEQUAL Function | Returns `true` if the first argument is greater than or equal to the second argument. Equivalent to the `>=` operator. |
| GREATERTHAN Function | Returns `true` if the first argument is greater than but not equal to the second argument. Equivalent to the `>` operator. |

In the town of Circleville, citizens are allowed to maintain a single crop circle in their backyard, as long as it confirms to the town regulations. Below is some data on the size of crop circles in town, with a separate entry for each home. Limits are displayed in the adjacent columns, with the `inclusive` columns indicating whether the minimum or maximum values are inclusive.

> **Tip:** As part of this exercise, you can see how to you can extend your recipe to perform some simple financial analysis of the data.

**Source:**

| Location | Radius_ft | minRadius_ft | minInclusive | maxRadius_ft | maxInclusive |
|---|---|---|---|---|---|
| House1 | 55.5 | 10 | Y | 25 | N |
| House2 | 12 | 10 | Y | 25 | N |

| | | | | | |
|---|---|---|---|---|---|
| House3 | 14.25 | 10 | Y | 25 | N |
| House4 | 3.5 | 10 | Y | 25 | N |
| House5 | 27 | 10 | Y | 25 | N |

**Transformation:**

After the data is loaded into the Transformer page, you can begin comparing column values:

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | LESSTHANEQUAL(Radius_ft,minRadius_ft) |
| Parameter: New column name | 'tooSmall' |

While accurate, the above transform does not account for the `minInclusive` value, which may be changed as part of your steps. Instead, you can delete the previous transform and use the following, which factors in the other column:

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | IF(minInclusive == 'Y',LESSTHANEQUAL(Radius_ft,minRadius_ft), LESSTHAN(Radius_ft,minRadius_ft)) |
| Parameter: New column name | 'tooSmall' |

In this case, the `IF` function tests whether the minimum value is inclusive (values of `10` are allowed). If so, the `LESSTHANEQUAL` function is applied. Otherwise, the `LESSTHAN` function is applied. For the maximum limit, the following step applies:

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | IF(maxInclusive == 'Y', GREATERTHANEQUAL(Radius_ft, maxRadius_ft),GREATERTHAN(Radius_ft,maxRadius_ft)) |
| Parameter: New column name | 'tooBig' |

Now, you can do some analysis of this data. First, you can insert a column containing the amount of the fine per foot above the maximum or below the minimum. Before the first `derive` command, insert the following, which is the fine ($15.00) for each foot above or below the limits:

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | 15 |

| Parameter: New column name | `'fineDollarsPerFt'` |
|---|---|

At the end of the recipe, add the following new line, which calculates the fine for crop circles that are too small:

| Transformation Name | `New formula` |
|---|---|
| Parameter: Formula type | `Single row formula` |
| Parameter: Formula | `IF(tooSmall == 'true', (minRadius_ft - Radius_ft) * fineDollarsPerFt, 0.0)` |
| Parameter: New column name | `'fine_Dollars'` |

The above captures the too-small violations. To also capture the too-big violations, change the above to the following:

| Transformation Name | `New formula` |
|---|---|
| Parameter: Formula type | `Single row formula` |
| Parameter: Formula | `IF(tooSmall == 'true', (minRadius_ft - Radius_ft) * fineDollarsPerFt, if(tooBig == 'true', (Radius_ft - maxRadius_ft) * fineDollarsPerFt, '0.0'))` |
| Parameter: New column name | `'fine_Dollars'` |

In place of the original "false" expression (`0.0`), the above adds the test for the too-big values, so that all fines are included in a single column. You can reformat the `fine_Dollars` column to be in dollar format:

| Transformation Name | `Edit column with formula` |
|---|---|
| Parameter: Columns | `fine_Dollars` |
| Parameter: Formula | `NUMFORMAT(fine_Dollars, '$###.00')` |

**Results:**

After you delete the columns used in the calculation and move the remaining ones, you should end up with a dataset similar to the following:

| Location | fineDollarsPerFt | Radius_ft | minRadius_ft | minInclusive | maxRadius_ft | maxInclusive | fineDollars |
|---|---|---|---|---|---|---|---|
| House1 | 15 | 55.5 | 10 | Y | 25 | N | $457.50 |
| House2 | 15 | 12 | 10 | Y | 25 | N | $0.00 |
| House3 | 15 | 14.25 | 10 | Y | 25 | N | $0.00 |
| House4 | 15 | 3.5 | 10 | Y | 25 | N | $97.50 |
| House5 | 15 | 27 | 10 | Y | 25 | N | $30.00 |

Now that you have created all of the computations for generating these values, you can change values for `minRadius_ft`, `maxRadius_ft`, and `fineDollarsPerFt` to analyze the resulting fine revenue. Before or after the transform where you set the value for `fineDollarsPerFt`, you can insert something like the following:

| Transformation Name | Edit column with formula |
|---|---|
| Parameter: Columns | minRadius_ft |
| Parameter: Formula | '12.5' |

After the step is added, select the last line in the recipe. Then, you can see how the values in the `fineDollars` column have been updated.

# GREATERTHANEQUAL Function

**Contents:**

- *Basic Usage*
- *Syntax and Arguments*
    - *value1, value2*
- *Examples*
    - *Example - Basic Comparison Functions*
    - *Example - Using Comparisons to Test Ranges*

---

Returns `true` if the first argument is greater than or equal to the second argument. Equivalent to the `>=` operator.

- Each argument can be a literal Integer or Decimal number, a function returning a number, or a reference to a column containing numbers.

Since the function returns a Boolean value, it can be used as a function or a conditional.

> **NOTE:** Within an expression, you might choose to use the corresponding operator, instead of this function. For more information, see *Comparison Operators*.

**Wrangle vs. SQL:** This function is part of Wrangle , a proprietary data transformation language. Wrangle is not SQL. For more information, see *Wrangle Language*.

## Basic Usage

```
greaterthanequal(myValue, minLimit)
```

**Output:** Returns `true` when the value in the `myValue` column is greater than or equal to the value in `minLimit`.

## Syntax and Arguments

```
greaterthanequal(value1, value2)
```

| Argument | Required? | Data Type | Description |
|----------|-----------|-----------|-------------|
| value1 | Y | string | The first value. This can be a number, a function returning a number, or a column containing numbers. |
| value2 | Y | string | The second value. This can be a number, a function returning a number, or a column containing numbers. |

For more information on syntax standards, see *Language Documentation Syntax Notes*.

**value1, value2**

Names of the column, expressions, or literals to compare.

- Missing values generate missing string results.

**Usage Notes:**

| Required? | Data Type | Example Value |
|---|---|---|
| Yes | Column reference, function, or numeric or String value | `myColumn` |

## Examples

> **Tip:** For additional examples, see *Common Tasks*.

### Example - Basic Comparison Functions

This example illustrates the comparison functions in Designer Cloud Powered by Trifacta® Enterprise Edition.

**Functions**:

| Item | Description |
|---|---|
| LESSTHAN Function | Returns `true` if the first argument is less than but not equal to the second argument. Equivalent to the < operator. |
| LESSTHANEQUAL Function | Returns `true` if the first argument is less than or equal to the second argument. Equivalent to the <= operator. |
| EQUAL Function | Returns `true` if the first argument is equal to the second argument. Equivalent to the = operator. |
| NOTEQUAL Function | Returns `true` if the first argument is not equal to the second argument. Equivalent to the <> or != operator. |
| GREATERTHAN Function | Returns `true` if the first argument is greater than but not equal to the second argument. Equivalent to the > operator. |
| GREATERTHANEQUAL Function | Returns `true` if the first argument is greater than or equal to the second argument. Equivalent to the >= operator. |

**Source:**

| colA | colB |
|---|---|
| 1 | 11 |
| 2 | 10 |
| 3 | 9 |
| 4 | 8 |
| 5 | 7 |
| 6 | 6 |
| 7 | 5 |
| 8 | 4 |
| 9 | 3 |
| 10 | 2 |

| 11 | 1 |
|----|---|

**Transformation:**

Add the following transforms to your recipe, one for each comparison function:

| | |
|---|---|
| **Transformation Name** | `New formula` |
| **Parameter: Formula type** | `Single row formula` |
| **Parameter: Formula** | `LESSTHAN(colA, colB)` |
| **Parameter: New column name** | `'lt'` |

| | |
|---|---|
| **Transformation Name** | `New formula` |
| **Parameter: Formula type** | `Single row formula` |
| **Parameter: Formula** | `LESSTHANEQUAL(colA, colB)` |
| **Parameter: New column name** | `'lte'` |

| | |
|---|---|
| **Transformation Name** | `New formula` |
| **Parameter: Formula type** | `Single row formula` |
| **Parameter: Formula** | `EQUAL(colA, colB)` |
| **Parameter: New column name** | `'eq'` |

| | |
|---|---|
| **Transformation Name** | `New formula` |
| **Parameter: Formula type** | `Single row formula` |
| **Parameter: Formula** | `NOTEQUAL(colA, colB)` |
| **Parameter: New column name** | `'neq'` |

| | |
|---|---|
| **Transformation Name** | `New formula` |
| **Parameter: Formula type** | `Single row formula` |
| **Parameter: Formula** | `GREATERTHAN(colA, colB)` |
| **Parameter: New column name** | `'gt'` |

| | |
|---|---|
| **Transformation Name** | `New formula` |
| **Parameter: Formula type** | `Single row formula` |
| **Parameter: Formula** | `GREATERTHANEQUAL(colA, colB)` |
| **Parameter: New column name** | `'gte'` |

**Results:**

| colA | colB | gte | gt | neq | eq | lte | lt |
|------|------|-------|-------|-------|-------|-------|-------|
| 1 | 11 | false | false | true | false | true | true |
| 2 | 10 | false | false | true | false | true | true |
| 3 | 9 | false | false | true | false | true | true |
| 4 | 8 | false | false | true | false | true | true |
| 5 | 7 | false | false | true | false | true | true |
| 6 | 6 | true | false | false | true | true | false |
| 7 | 5 | true | true | true | false | false | false |
| 8 | 4 | true | true | true | false | false | false |
| 9 | 3 | true | true | true | false | false | false |
| 10 | 2 | true | true | true | false | false | false |
| 11 | 1 | true | true | true | false | false | false |

## Example - Using Comparisons to Test Ranges

This example demonstrates functions for comparing the relative values of two functions.

**Functions:**

| Item | Description |
|------|-------------|
| LESSTHANEQUAL Function | Returns `true` if the first argument is less than or equal to the second argument. Equivalent to the `<=` operator. |
| LESSTHAN Function | Returns `true` if the first argument is less than but not equal to the second argument. Equivalent to the `<` operator. |
| GREATERTHANEQUAL Function | Returns `true` if the first argument is greater than or equal to the second argument. Equivalent to the `>=` operator. |
| GREATERTHAN Function | Returns `true` if the first argument is greater than but not equal to the second argument. Equivalent to the `>` operator. |

In the town of Circleville, citizens are allowed to maintain a single crop circle in their backyard, as long as it confirms to the town regulations. Below is some data on the size of crop circles in town, with a separate entry for each home. Limits are displayed in the adjacent columns, with the `inclusive` columns indicating whether the minimum or maximum values are inclusive.

> **Tip:** As part of this exercise, you can see how to you can extend your recipe to perform some simple financial analysis of the data.

**Source:**

| Location | Radius_ft | minRadius_ft | minInclusive | maxRadius_ft | maxInclusive |
|----------|-----------|--------------|--------------|--------------|--------------|
| House1 | 55.5 | 10 | Y | 25 | N |
| House2 | 12 | 10 | Y | 25 | N |
| House3 | 14.25 | 10 | Y | 25 | N |
| House4 | 3.5 | 10 | Y | 25 | N |

| House5 | 27 | 10 | Y | 25 | N |

**Transformation:**

After the data is loaded into the Transformer page, you can begin comparing column values:

| Transformation Name | `New formula` |
|---|---|
| **Parameter: Formula type** | `Single row formula` |
| **Parameter: Formula** | `LESSTHANEQUAL(Radius_ft,minRadius_ft)` |
| **Parameter: New column name** | `'tooSmall'` |

While accurate, the above transform does not account for the `minInclusive` value, which may be changed as part of your steps. Instead, you can delete the previous transform and use the following, which factors in the other column:

| Transformation Name | `New formula` |
|---|---|
| **Parameter: Formula type** | `Single row formula` |
| **Parameter: Formula** | `IF(minInclusive == 'Y',LESSTHANEQUAL(Radius_ft,minRadius_ft),`<br>`LESSTHAN(Radius_ft,minRadius_ft))` |
| **Parameter: New column name** | `'tooSmall'` |

In this case, the `IF` function tests whether the minimum value is inclusive (values of `10` are allowed). If so, the `LESSTHANEQUAL` function is applied. Otherwise, the `LESSTHAN` function is applied. For the maximum limit, the following step applies:

| Transformation Name | `New formula` |
|---|---|
| **Parameter: Formula type** | `Single row formula` |
| **Parameter: Formula** | `IF(maxInclusive == 'Y', GREATERTHANEQUAL(Radius_ft,`<br>`maxRadius_ft),GREATERTHAN(Radius_ft,maxRadius_ft))` |
| **Parameter: New column name** | `'tooBig'` |

Now, you can do some analysis of this data. First, you can insert a column containing the amount of the fine per foot above the maximum or below the minimum. Before the first `derive` command, insert the following, which is the fine (`$15.00`) for each foot above or below the limits:

| Transformation Name | `New formula` |
|---|---|
| **Parameter: Formula type** | `Single row formula` |
| **Parameter: Formula** | `15` |
| **Parameter: New column name** | `'fineDollarsPerFt'` |

At the end of the recipe, add the following new line, which calculates the fine for crop circles that are too small:

| | |
|---|---|
| Transformation Name | `New formula` |
| Parameter: Formula type | `Single row formula` |
| Parameter: Formula | `IF(tooSmall == 'true', (minRadius_ft - Radius_ft) *`<br>`fineDollarsPerFt, 0.0)` |
| Parameter: New column name | `'fine_Dollars'` |

The above captures the too-small violations. To also capture the too-big violations, change the above to the following:

| | |
|---|---|
| Transformation Name | `New formula` |
| Parameter: Formula type | `Single row formula` |
| Parameter: Formula | `IF(tooSmall == 'true', (minRadius_ft - Radius_ft) *`<br>`fineDollarsPerFt, if(tooBig == 'true', (Radius_ft -`<br>`maxRadius_ft) * fineDollarsPerFt, '0.0'))` |
| Parameter: New column name | `'fine_Dollars'` |

In place of the original "false" expression (`0.0`), the above adds the test for the too-big values, so that all fines are included in a single column. You can reformat the `fine_Dollars` column to be in dollar format:

| | |
|---|---|
| Transformation Name | `Edit column with formula` |
| Parameter: Columns | `fine_Dollars` |
| Parameter: Formula | `NUMFORMAT(fine_Dollars, '$###.00')` |

**Results:**

After you delete the columns used in the calculation and move the remaining ones, you should end up with a dataset similar to the following:

| Location | fineDollarsPerFt | Radius_ft | minRadius_ft | minInclusive | maxRadius_ft | maxInclusive | fineDollars |
|---|---|---|---|---|---|---|---|
| House1 | 15 | 55.5 | 10 | Y | 25 | N | $457.50 |
| House2 | 15 | 12 | 10 | Y | 25 | N | $0.00 |
| House3 | 15 | 14.25 | 10 | Y | 25 | N | $0.00 |
| House4 | 15 | 3.5 | 10 | Y | 25 | N | $97.50 |
| House5 | 15 | 27 | 10 | Y | 25 | N | $30.00 |

Now that you have created all of the computations for generating these values, you can change values for `minRa dius_ft`, `maxRadius_ft`, and `fineDollarsPerFt` to analyze the resulting fine revenue. Before or after the transform where you set the value for `fineDollarsPerFt`, you can insert something like the following:

| | |
|---|---|
| **Transformation Name** | Edit column with formula |
| **Parameter: Columns** | minRadius_ft |
| **Parameter: Formula** | '12.5' |

After the step is added, select the last line in the recipe. Then, you can see how the values in the `fineDollars` column have been updated.

# LESSTHAN Function

**Contents:**

---

Returns `true` if the first argument is less than but not equal to the second argument. Equivalent to the `<` operator.

- Each argument can be a literal Integer or Decimal number, a function returning a number, or a reference to a column containing numbers.

Since the function returns a Boolean value, it can be used as a function or a conditional.

> **NOTE:** Within an expression, you might choose to use the corresponding operator, instead of this function. For more information, see *Comparison Operators*.

**Wrangle vs. SQL:** This function is part of Wrangle , a proprietary data transformation language. Wrangle is not SQL. For more information, see *Wrangle Language*.

## Basic Usage

```
lessthan(Errors, 10)
```

**Output:** Returns `true` when the value in the `Errors` column is less than 10.

## Syntax and Arguments

```
lessthan(value1, value2)
```

| Argument | Required? | Data Type | Description |
|----------|-----------|-----------|-------------|
| value1 | Y | string | The first value. This can be a number, a function returning a number, or a column containing numbers. |
| value2 | Y | string | The second value. This can be a number, a function returning a number, or a column containing numbers. |

For more information on syntax standards, see *Language Documentation Syntax Notes*.

**value1, value2**

Names of the column, expressions, or literals to compare.

- Missing values generate missing string results.

**Usage Notes:**

| Required? | Data Type | Example Value |
|---|---|---|
| Yes | Column reference, function, or numeric or String value | `myColumn` |

## Examples

> **Tip:** For additional examples, see *Common Tasks*.

**Example - Basic Comparison Functions**

This example illustrates the comparison functions in Designer Cloud Powered by Trifacta® Enterprise Edition.

**Functions**:

| Item | Description |
|---|---|
| LESSTHAN Function | Returns `true` if the first argument is less than but not equal to the second argument. Equivalent to the < operator. |
| LESSTHANEQUAL Function | Returns `true` if the first argument is less than or equal to the second argument. Equivalent to the <= operator. |
| EQUAL Function | Returns `true` if the first argument is equal to the second argument. Equivalent to the = operator. |
| NOTEQUAL Function | Returns `true` if the first argument is not equal to the second argument. Equivalent to the <> or != operator. |
| GREATERTHAN Function | Returns `true` if the first argument is greater than but not equal to the second argument. Equivalent to the > operator. |
| GREATERTHANEQUAL Function | Returns `true` if the first argument is greater than or equal to the second argument. Equivalent to the >= operator. |

**Source:**

| colA | colB |
|---|---|
| 1 | 11 |
| 2 | 10 |
| 3 | 9 |
| 4 | 8 |
| 5 | 7 |
| 6 | 6 |
| 7 | 5 |
| 8 | 4 |
| 9 | 3 |
| 10 | 2 |

| 11 | 1 |
|----|---|

**Transformation:**

Add the following transforms to your recipe, one for each comparison function:

| Transformation Name | New formula |
|---|---|
| **Parameter: Formula type** | Single row formula |
| **Parameter: Formula** | LESSTHAN(colA, colB) |
| **Parameter: New column name** | 'lt' |

| Transformation Name | New formula |
|---|---|
| **Parameter: Formula type** | Single row formula |
| **Parameter: Formula** | LESSTHANEQUAL(colA, colB) |
| **Parameter: New column name** | 'lte' |

| Transformation Name | New formula |
|---|---|
| **Parameter: Formula type** | Single row formula |
| **Parameter: Formula** | EQUAL(colA, colB) |
| **Parameter: New column name** | 'eq' |

| Transformation Name | New formula |
|---|---|
| **Parameter: Formula type** | Single row formula |
| **Parameter: Formula** | NOTEQUAL(colA, colB) |
| **Parameter: New column name** | 'neq' |

| Transformation Name | New formula |
|---|---|
| **Parameter: Formula type** | Single row formula |
| **Parameter: Formula** | GREATERTHAN(colA, colB) |
| **Parameter: New column name** | 'gt' |

| Transformation Name | New formula |
|---|---|
| **Parameter: Formula type** | Single row formula |
| **Parameter: Formula** | GREATERTHANEQUAL(colA, colB) |
| **Parameter: New column name** | 'gte' |

**Results:**

| colA | colB | gte | gt | neq | eq | lte | lt |
|------|------|-----|----|----|----|-----|-----|
| 1 | 11 | false | false | true | false | true | true |
| 2 | 10 | false | false | true | false | true | true |
| 3 | 9 | false | false | true | false | true | true |
| 4 | 8 | false | false | true | false | true | true |
| 5 | 7 | false | false | true | false | true | true |
| 6 | 6 | true | false | false | true | true | false |
| 7 | 5 | true | true | true | false | false | false |
| 8 | 4 | true | true | true | false | false | false |
| 9 | 3 | true | true | true | false | false | false |
| 10 | 2 | true | true | true | false | false | false |
| 11 | 1 | true | true | true | false | false | false |

## Example - Using Comparisons to Test Ranges

This example demonstrates functions for comparing the relative values of two functions.

**Functions:**

| Item | Description |
|------|-------------|
| LESSTHANEQUAL Function | Returns `true` if the first argument is less than or equal to the second argument. Equivalent to the `<=` operator. |
| LESSTHAN Function | Returns `true` if the first argument is less than but not equal to the second argument. Equivalent to the `<` operator. |
| GREATERTHANEQUAL Function | Returns `true` if the first argument is greater than or equal to the second argument. Equivalent to the `>=` operator. |
| GREATERTHAN Function | Returns `true` if the first argument is greater than but not equal to the second argument. Equivalent to the `>` operator. |

In the town of Circleville, citizens are allowed to maintain a single crop circle in their backyard, as long as it confirms to the town regulations. Below is some data on the size of crop circles in town, with a separate entry for each home. Limits are displayed in the adjacent columns, with the `inclusive` columns indicating whether the minimum or maximum values are inclusive.

> **Tip:** As part of this exercise, you can see how to you can extend your recipe to perform some simple financial analysis of the data.

**Source:**

| Location | Radius_ft | minRadius_ft | minInclusive | maxRadius_ft | maxInclusive |
|----------|-----------|--------------|--------------|--------------|--------------|
| House1 | 55.5 | 10 | Y | 25 | N |
| House2 | 12 | 10 | Y | 25 | N |
| House3 | 14.25 | 10 | Y | 25 | N |
| House4 | 3.5 | 10 | Y | 25 | N |

| House5 | 27 | 10 | Y | 25 | N |

**Transformation:**

After the data is loaded into the Transformer page, you can begin comparing column values:

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | LESSTHANEQUAL(Radius_ft,minRadius_ft) |
| Parameter: New column name | 'tooSmall' |

While accurate, the above transform does not account for the `minInclusive` value, which may be changed as part of your steps. Instead, you can delete the previous transform and use the following, which factors in the other column:

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | IF(minInclusive == 'Y',LESSTHANEQUAL(Radius_ft,minRadius_ft), LESSTHAN(Radius_ft,minRadius_ft)) |
| Parameter: New column name | 'tooSmall' |

In this case, the `IF` function tests whether the minimum value is inclusive (values of `10` are allowed). If so, the `LESSTHANEQUAL` function is applied. Otherwise, the `LESSTHAN` function is applied. For the maximum limit, the following step applies:

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | IF(maxInclusive == 'Y', GREATERTHANEQUAL(Radius_ft, maxRadius_ft),GREATERTHAN(Radius_ft,maxRadius_ft)) |
| Parameter: New column name | 'tooBig' |

Now, you can do some analysis of this data. First, you can insert a column containing the amount of the fine per foot above the maximum or below the minimum. Before the first `derive` command, insert the following, which is the fine (`$15.00`) for each foot above or below the limits:

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | 15 |
| Parameter: New column name | 'fineDollarsPerFt' |

At the end of the recipe, add the following new line, which calculates the fine for crop circles that are too small:

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | IF(tooSmall == 'true', (minRadius_ft - Radius_ft) * fineDollarsPerFt, 0.0) |
| Parameter: New column name | 'fine_Dollars' |

The above captures the too-small violations. To also capture the too-big violations, change the above to the following:

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | IF(tooSmall == 'true', (minRadius_ft - Radius_ft) * fineDollarsPerFt, if(tooBig == 'true', (Radius_ft - maxRadius_ft) * fineDollarsPerFt, '0.0')) |
| Parameter: New column name | 'fine_Dollars' |

In place of the original "false" expression (`0.0`), the above adds the test for the too-big values, so that all fines are included in a single column. You can reformat the `fine_Dollars` column to be in dollar format:

| Transformation Name | Edit column with formula |
|---|---|
| Parameter: Columns | fine_Dollars |
| Parameter: Formula | NUMFORMAT(fine_Dollars, '$###.00') |

**Results:**

After you delete the columns used in the calculation and move the remaining ones, you should end up with a dataset similar to the following:

| Location | fineDollarsPerFt | Radius_ft | minRadius_ft | minInclusive | maxRadius_ft | maxInclusive | fineDollars |
|---|---|---|---|---|---|---|---|
| House1 | 15 | 55.5 | 10 | Y | 25 | N | $457.50 |
| House2 | 15 | 12 | 10 | Y | 25 | N | $0.00 |
| House3 | 15 | 14.25 | 10 | Y | 25 | N | $0.00 |
| House4 | 15 | 3.5 | 10 | Y | 25 | N | $97.50 |
| House5 | 15 | 27 | 10 | Y | 25 | N | $30.00 |

Now that you have created all of the computations for generating these values, you can change values for `minRadius_ft`, `maxRadius_ft`, and `fineDollarsPerFt` to analyze the resulting fine revenue. Before or after the transform where you set the value for `fineDollarsPerFt`, you can insert something like the following:

| | |
|---|---|
| **Transformation Name** | Edit column with formula |
| **Parameter: Columns** | minRadius_ft |
| **Parameter: Formula** | '12.5' |

After the step is added, select the last line in the recipe. Then, you can see how the values in the `fineDollars` column have been updated.

# LESSTHANEQUAL Function

**Contents:**

- *Basic Usage*
- *Syntax and Arguments*
  - *value1, value2*
- *Examples*
  - *Example - Basic Comparison Functions*
  - *Example - Using Comparisons to Test Ranges*

---

Returns `true` if the first argument is less than or equal to the second argument. Equivalent to the `<=` operator.

- Each argument can be a literal Integer or Decimal number, a function returning a number, or a reference to a column containing numbers.

Since the function returns a Boolean value, it can be used as a function or a conditional.

> **NOTE:** Within an expression, you might choose to use the corresponding operator, instead of this function. For more information, see *Comparison Operators*.

**Wrangle vs. SQL:** This function is part of Wrangle , a proprietary data transformation language. Wrangle is not SQL. For more information, see *Wrangle Language*.

## Basic Usage

```
lessthanequal(myValue, maxLimit)
```

**Output:** Returns `true` when the the the `myValue` column is less than or equal to the value in `maxLimit`.

## Syntax and Arguments

```
lessthanequal(value1, value2)
```

| Argument | Required? | Data Type | Description |
|----------|-----------|-----------|-------------|
| value1 | Y | string | The first value. This can be a number, a function returning a number, or a column containing numbers. |
| value2 | Y | string | The second value. This can be a number, a function returning a number, or a column containing numbers. |

For more information on syntax standards, see *Language Documentation Syntax Notes*.

**value1, value2**

Names of the column, expressions, or literals to compare.

- Missing values generate missing string results.

**Usage Notes:**

| Required? | Data Type | Example Value |
|-----------|-----------|---------------|
| Yes | Column reference, function, or numeric or String value | `myColumn` |

## Examples

> **Tip:** For additional examples, see *Common Tasks*.

### Example - Basic Comparison Functions

This example illustrates the comparison functions in Designer Cloud Powered by Trifacta® Enterprise Edition.

**Functions**:

| Item | Description |
|------|-------------|
| LESSTHAN Function | Returns `true` if the first argument is less than but not equal to the second argument. Equivalent to the `<` operator. |
| LESSTHANEQUAL Function | Returns `true` if the first argument is less than or equal to the second argument. Equivalent to the `<=` operator. |
| EQUAL Function | Returns `true` if the first argument is equal to the second argument. Equivalent to the `=` operator. |
| NOTEQUAL Function | Returns `true` if the first argument is not equal to the second argument. Equivalent to the `<>` or `!=` operator. |
| GREATERTHAN Function | Returns `true` if the first argument is greater than but not equal to the second argument. Equivalent to the `>` operator. |
| GREATERTHANEQUAL Function | Returns `true` if the first argument is greater than or equal to the second argument. Equivalent to the `>=` operator. |

**Source:**

| colA | colB |
|------|------|
| 1 | 11 |
| 2 | 10 |
| 3 | 9 |
| 4 | 8 |
| 5 | 7 |
| 6 | 6 |
| 7 | 5 |
| 8 | 4 |
| 9 | 3 |
| 10 | 2 |
| 11 | 1 |

**Transformation:**

Add the following transforms to your recipe, one for each comparison function:

| Transformation Name | New formula |
|---|---|
| **Parameter: Formula type** | Single row formula |
| **Parameter: Formula** | LESSTHAN(colA, colB) |
| **Parameter: New column name** | 'lt' |

| Transformation Name | New formula |
|---|---|
| **Parameter: Formula type** | Single row formula |
| **Parameter: Formula** | LESSTHANEQUAL(colA, colB) |
| **Parameter: New column name** | 'lte' |

| Transformation Name | New formula |
|---|---|
| **Parameter: Formula type** | Single row formula |
| **Parameter: Formula** | EQUAL(colA, colB) |
| **Parameter: New column name** | 'eq' |

| Transformation Name | New formula |
|---|---|
| **Parameter: Formula type** | Single row formula |
| **Parameter: Formula** | NOTEQUAL(colA, colB) |
| **Parameter: New column name** | 'neq' |

| Transformation Name | New formula |
|---|---|
| **Parameter: Formula type** | Single row formula |
| **Parameter: Formula** | GREATERTHAN(colA, colB) |
| **Parameter: New column name** | 'gt' |

| Transformation Name | New formula |
|---|---|
| **Parameter: Formula type** | Single row formula |
| **Parameter: Formula** | GREATERTHANEQUAL(colA, colB) |
| **Parameter: New column name** | 'gte' |

**Results:**

| colA | colB | gte | gt | neq | eq | lte | lt |
|---|---|---|---|---|---|---|---|
| 1 | 11 | false | false | true | false | true | true |
| 2 | 10 | false | false | true | false | true | true |
| 3 | 9 | false | false | true | false | true | true |
| 4 | 8 | false | false | true | false | true | true |
| 5 | 7 | false | false | true | false | true | true |
| 6 | 6 | true | false | false | true | true | false |
| 7 | 5 | true | true | true | false | false | false |
| 8 | 4 | true | true | true | false | false | false |
| 9 | 3 | true | true | true | false | false | false |
| 10 | 2 | true | true | true | false | false | false |
| 11 | 1 | true | true | true | false | false | false |

## Example - Using Comparisons to Test Ranges

This example demonstrates functions for comparing the relative values of two functions.

### Functions:

| Item | Description |
|---|---|
| LESSTHANEQUAL Function | Returns `true` if the first argument is less than or equal to the second argument. Equivalent to the <= operator. |
| LESSTHAN Function | Returns `true` if the first argument is less than but not equal to the second argument. Equivalent to the < operator. |
| GREATERTHANEQUAL Function | Returns `true` if the first argument is greater than or equal to the second argument. Equivalent to the >= operator. |
| GREATERTHAN Function | Returns `true` if the first argument is greater than but not equal to the second argument. Equivalent to the > operator. |

In the town of Circleville, citizens are allowed to maintain a single crop circle in their backyard, as long as it confirms to the town regulations. Below is some data on the size of crop circles in town, with a separate entry for each home. Limits are displayed in the adjacent columns, with the `inclusive` columns indicating whether the minimum or maximum values are inclusive.

> **Tip:** As part of this exercise, you can see how to you can extend your recipe to perform some simple financial analysis of the data.

### Source:

| Location | Radius_ft | minRadius_ft | minInclusive | maxRadius_ft | maxInclusive |
|---|---|---|---|---|---|
| House1 | 55.5 | 10 | Y | 25 | N |
| House2 | 12 | 10 | Y | 25 | N |
| House3 | 14.25 | 10 | Y | 25 | N |
| House4 | 3.5 | 10 | Y | 25 | N |
| House5 | 27 | 10 | Y | 25 | N |

**Transformation:**

After the data is loaded into the Transformer page, you can begin comparing column values:

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | LESSTHANEQUAL(Radius_ft,minRadius_ft) |
| Parameter: New column name | 'tooSmall' |

While accurate, the above transform does not account for the `minInclusive` value, which may be changed as part of your steps. Instead, you can delete the previous transform and use the following, which factors in the other column:

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | IF(minInclusive == 'Y',LESSTHANEQUAL(Radius_ft,minRadius_ft), LESSTHAN(Radius_ft,minRadius_ft)) |
| Parameter: New column name | 'tooSmall' |

In this case, the `IF` function tests whether the minimum value is inclusive (values of `10` are allowed). If so, the `LESSTHANEQUAL` function is applied. Otherwise, the `LESSTHAN` function is applied. For the maximum limit, the following step applies:

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | IF(maxInclusive == 'Y', GREATERTHANEQUAL(Radius_ft, maxRadius_ft),GREATERTHAN(Radius_ft,maxRadius_ft)) |
| Parameter: New column name | 'tooBig' |

Now, you can do some analysis of this data. First, you can insert a column containing the amount of the fine per foot above the maximum or below the minimum. Before the first `derive` command, insert the following, which is the fine ($15.00) for each foot above or below the limits:

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | 15 |
| Parameter: New column name | 'fineDollarsPerFt' |

At the end of the recipe, add the following new line, which calculates the fine for crop circles that are too small:

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | `Single row formula` |
| Parameter: Formula | `IF(tooSmall == 'true', (minRadius_ft - Radius_ft) *`<br>`fineDollarsPerFt, 0.0)` |
| Parameter: New column name | `'fine_Dollars'` |

The above captures the too-small violations. To also capture the too-big violations, change the above to the following:

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | `Single row formula` |
| Parameter: Formula | `IF(tooSmall == 'true', (minRadius_ft - Radius_ft) *`<br>`fineDollarsPerFt, if(tooBig == 'true', (Radius_ft -`<br>`maxRadius_ft) * fineDollarsPerFt, '0.0'))` |
| Parameter: New column name | `'fine_Dollars'` |

In place of the original "false" expression (`0.0`), the above adds the test for the too-big values, so that all fines are included in a single column. You can reformat the `fine_Dollars` column to be in dollar format:

| Transformation Name | Edit column with formula |
|---|---|
| Parameter: Columns | `fine_Dollars` |
| Parameter: Formula | `NUMFORMAT(fine_Dollars, '$###.00')` |

**Results:**

After you delete the columns used in the calculation and move the remaining ones, you should end up with a dataset similar to the following:

| Location | fineDollarsPerFt | Radius_ft | minRadius_ft | minInclusive | maxRadius_ft | maxInclusive | fineDollars |
|---|---|---|---|---|---|---|---|
| House1 | 15 | 55.5 | 10 | Y | 25 | N | $457.50 |
| House2 | 15 | 12 | 10 | Y | 25 | N | $0.00 |
| House3 | 15 | 14.25 | 10 | Y | 25 | N | $0.00 |
| House4 | 15 | 3.5 | 10 | Y | 25 | N | $97.50 |
| House5 | 15 | 27 | 10 | Y | 25 | N | $30.00 |

Now that you have created all of the computations for generating these values, you can change values for `minRadius_ft`, `maxRadius_ft`, and `fineDollarsPerFt` to analyze the resulting fine revenue. Before or after the transform where you set the value for `fineDollarsPerFt`, you can insert something like the following:

| Transformation Name | Edit column with formula |
|---|---|
| Parameter: Columns | `minRadius_ft` |

| Parameter: Formula | '12.5' |
| --- | --- |

After the step is added, select the last line in the recipe. Then, you can see how the values in the `fineDollars` column have been updated.

# Math Functions

These mathematical functions can be applied to your Wrangle transformations. These functions are typically inserted in the `value` parameter as part of the transform.

A function can require zero or more arguments.

# Numeric Operators

**Contents:**

---

Numeric operators enable you to generate new values based on a computation (e.g. `3 + 4`).

For each expression, the numeric operator is applied from left to right:

```
(left-hand side) (operator) (right-hand side)
```

These evaluations result in a numeric output, which can be an Integer or Decimal depending on the input values. The following operators are supported:

| Operator Name | Symbol | Example Expression | Output | Notes |
|---|---|---|---|---|
| add | + | `3 + 6` | `9` | |
| subtract | – | `6 – 5` | `1` | |
| multiply | * | `3 * 6` | `18` | |
| divide | / | `25 / 5` | `5` | |
| modulo | % | `5 % 4` | `1` | Computes the remainder as an integer of the first parameter divided by the second parameter. Input parameters must be Integers, column references to Integers, or an expression that evaluates to an Integer. |
| power | `pow` | `pow(4,3)` | `64` | Power is implemented as a function. see *POW Function*. |
| negate | – | `– myColumn` | opposite of the value in `myColumn` | This operator supports only one operand. Parenthetical references are supported. See the example below. |

The above examples apply to integer values only. Below, you can review how the comparison operators apply to different data types.

## Usage

Numeric operators are used to perform numeric manipulations on columns of data, Integer or Decimal constants, or both. Typically, they are applied in evaluations of values or rows.

Example data:

| X | Y |
|---|---|
| 2 | 1 |
| 6 | 4 |
| 7 | 10 |
| 10 | 0 |

**Transformations:**

| | |
|---|---|
| **Transformation Name** | `New formula` |
| **Parameter: Formula type** | `Single row formula` |
| **Parameter: Formula** | `(X + Y)` |
| **Parameter: New column name** | `'add'` |

| | |
|---|---|
| **Transformation Name** | `New formula` |
| **Parameter: Formula type** | `Single row formula` |
| **Parameter: Formula** | `(X - Y)` |
| **Parameter: New column name** | `'subtract'` |

| | |
|---|---|
| **Transformation Name** | `New formula` |
| **Parameter: Formula type** | `Single row formula` |
| **Parameter: Formula** | `(X * Y)` |
| **Parameter: New column name** | `'multiply'` |

| | |
|---|---|
| **Transformation Name** | `New formula` |
| **Parameter: Formula type** | `Single row formula` |
| **Parameter: Formula** | `(X / Y)` |
| **Parameter: New column name** | `'divide'` |

| | |
|---|---|
| **Transformation Name** | `New formula` |
| **Parameter: Formula type** | `Single row formula` |
| **Parameter: Formula** | `(X % Y)` |
| **Parameter: New column name** | `'modulo'` |

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | `Single row formula` |
| Parameter: Formula | `pow(X,Y)` |
| Parameter: New column name | `'power'` |

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | `Single row formula` |
| Parameter: Formula | `-(X-Y)` |
| Parameter: New column name | `'negativeXminusY'` |

**Results:**

Your output looks like the following. Below, `(null value)` indicates that a null value is generated for the computation.

| X | Y | add | subtract | multiply | divide | modulo | power | negativeXminusY |
|---|---|---|---|---|---|---|---|---|
| 2 | 1 | 3 | 1 | 2 | 2 | 0 | 2 | -1 |
| 6 | 4 | 10 | 2 | 24 | 1.5 | 2 | 1296 | -3 |
| 7 | 10 | 17 | -3 | 70 | 0.7 | 7 | 282475249 | 3 |
| 10 | 0 | 20 | 10 | 0 | (null value) | (null value) | 1 | -10 |

## Examples

> **Tip:** For additional examples, see *Common Tasks*.

> **NOTE:** When a numeric operator is applied to a set of values, the type of data of each source value is re-inferred to match any literal values used on the other side of the expression. This method allows for more powerful comparisons.
>
> In the following examples, values taken from the `MySource` column are re-typed to match the inferred data type of the other side of the expression.

**add**

| Column Type | Example Transformation | Output | Notes |
|---|---|---|---|

| Integer/Decimal | | | Generated values are sum of values in `MySource` column and the constant (5 or 2.5). | |
|---|---|---|---|---|
| | **Transformation Name** | `New formula` | | |
| | **Parameter: Formula type** | `Single row formula` | | |
| | **Parameter: Formula** | `(MySource + 5)` | | |
| Datetime | You cannot directly add Datetime values. You can use the `DATEDIF` function to generate the number of days difference between two date values. See *DATEDIF Function*. | | | |
| String | You cannot add strings together.<br><br>• You can use the `MERGE` transform to concatenate columns of string values together. See *Merge Transform*.<br>• You can use the `ARRAYCONCAT` function to concatenate multiple columns of array type together. See *ARRAYCONCAT Function*. | | | For computational purposes, all data types not previously listed in this table behave like strings. |

### subtact

| Column Type | Example Transformation | | Output | Notes |
|---|---|---|---|---|
| Integer/Decimal | | | Generated values are difference between values in `MySource` column and the constant (5 or 2.5). | |
| | **Transformation Name** | `New formula` | | |
| | **Parameter: Formula type** | `Single row formula` | | |
| | **Parameter: Formula** | `(MySource - 5)` | | |
| | **Transformation Name** | `New formula` | | |
| | **Parameter: Formula type** | `Single row formula` | | |
| | **Parameter: Formula** | `(MySource - 2.5)` | | |
| Datetime | You cannot directly subtract Datetime values. You must use the `DATEDIF` function to generate the number of days difference between two date values. See *DATEDIF Function*. | | | |
| String | You cannot differentiate strings directly.<br><br>• You can use the `SUBSTRING` function to locate one string within the other. If found, this function returns the index of the value in the source string where the substring is located. This index value can used as an input to the `LEFT` and `RIGHT` functions to remove the substring. See *SUBSTRING Function*. | | | For computational purposes, all data types not previously listed in this table behave like strings. |

## multiply

| Column Type | Example Transformation | | Output | Notes |
|---|---|---|---|---|
| Integer/Decimal | **Transformation Name** | `New formula` | Generated values are the product of values in the `MySource` column and the constant (5 or 2.5). | |
| | **Parameter: Formula type** | `Single row formula` | | |
| | **Parameter: Formula** | `(MySource * 5)` | | |
| | **Transformation Name** | `New formula` | | |
| | **Parameter: Formula type** | `Single row formula` | | |
| | **Parameter: Formula** | `(MySource * 2.5)` | | |
| Datetime | N/A | | | |
| String | N/A | | | |

## divide

| Column Type | Example Transformation | | Output | Notes |
|---|---|---|---|---|
| Integer /Decimal | **Transformation Name** | `New formula` | Generated values are the values in the `MySource` column divided by the constant (5 or 2.5). | |
| | **Parameter: Formula type** | `Single row formula` | | |
| | **Parameter: Formula** | `(MySource / 5)` | | |
| | **Transformation Name** | `New formula` | | |
| | **Parameter: Formula type** | `Single row formula` | | |
| | **Parameter: Formula** | `(MySource / 2.5)` | | |
| Datetime | N/A | | | |
| String | N/A | | | |

**modulo**

| Column Type | Example Transformation | | Output | Notes |
|---|---|---|---|---|
| Integer | **Transformation Name** | New formula | Generated values are the values in the `MySource` column divided by the constant (5 or 2.5). | |
| | **Parameter: Formula type** | Single row formula | | |
| | **Parameter: Formula** | (MySource % 5) | | |
| Decimal | **Transformation Name** | New formula | Not supported. Inputs must be of Integer type. | |
| | **Parameter: Formula type** | Single row formula | | |
| | **Parameter: Formula** | (MySource % 2.5) | | |
| Datetime | N/A | | | |
| String | N/A | | | |

# NUMFORMAT Function

**Contents:**

---

Formats a numeric set of values according to the specified number formatting. Source values can be a literal numeric value, a function returning a numeric value, or reference to a column containing an Integer or Decimal values.

- If the source value does not include a valid input for this function, a missing value is returned.
- When this function is applied, the column can be re-typed to a different data type. For example, if your format string (second parameter) is `'#'` (a single hash mark), then all values are rounded to the nearest integer, and the column is re-typed as Integer.

> **Tip:** In general, you should format your numeric data after you have completed your computations on it. In some cases, you might lose numeric precision in converting formats, or your data can be re-typed to a different data type (For example, Decimal to Integer).

- You can also use decimal separators and grouping separators when working with data from multiple locales. If no separators are provided, the U.S. format separators are used.
- When this function is applied to any column, the resulting column is of String type, so arithmetic operations are not possible on the resulting column.

> **NOTE:** If the function is unable to process the value, a null value is returned on Trifacta Photon. On other running environments, trailing characters that do not apply to numeric values or their formatting are simply dropped.

Designer Cloud Powered by Trifacta® Enterprise Edition supports a wide variety of number formats, following Java standards. For more information, please see Java's documentation.

> **NOTE:** This function just changes how the underlying cell value is displayed. If you round the value to a specific level of precision, please use the ROUND function. See *ROUND Function*.

For more information on formatting date values, see *DATEFORMAT Function*.

**Wrangle vs. SQL:** This function is part of Wrangle , a proprietary data transformation language. Wrangle is not SQL. For more information, see *Wrangle Language*.

## Basic Usage

```
Numformat(<code>MyPrice</code>, &apos;<code><span class="author-d-
1gg9uz65z1iz85zgdz68zmqkz84zo2qowz82zfhz88zz69zyz82z4z74z55g8rhz76zh7z67zk5z79zg5u1s77pz89z
h-ldquo">$ </span><span class="author-d-
1gg9uz65z1iz85zgdz68zmqkz84zo2qowz82zfhz88zz69zyz82z4z74z55g8rhz76zh7z67zk5z79zg5u1s77pz89z
">##,##.#</span></code>&apos;, &apos;<code>,</code>&apos; ,&apos;<code>.</code>&apos;)
```

**Output:** Returns the values from the `MyPrice` column by formatting the values using the specified formatting string and group and separators separators. For example, if the MyPrice column has a value of 12345.12 then it can be reformatted to $ 1,23,45.12 by using the above parameters.

## Syntax and Arguments

```
<code class="listtype-code listindent1 list-code1 lang-bash"><span>Numformat</span>
(<span>numeric_val</span>, <span>number_format_string</span>, [grouping_separator],
[decimal_separator])</code>
```

| Argument | Required? | Data Type | Description |
|---|---|---|---|
| numeric_val | Y | string, integer, or decimal | Literal value, function returning a numeric value, or name of Integer or Decimal column whose values are to be formatted |
| number_format_string | Y | string | Literal value of the number formatting string used to indicate location of separators, number of required digits, currency, percentage, and sign. |
| grouping_separator | N | string | A grouping representing grouping separator. By default, comma (,) is used as the grouping separator. |
| decimal_separator | N | string | A string representing decimal separator. By default, period (.) is used as the decimal separator. |

For more information on syntax standards, see *Language Documentation Syntax Notes*.

### numeric_val

Literal numeric value, a function that returns a numeric value, or the name of the column whose Integer or Decimal data is to be formatted.

- Values with more than 20 digits after the Decimal point are truncated by this function.
- Missing values for this function in the source data result in missing values in the output.
- Multiple columns and wildcards are not supported.
- Using a dash as a negative value indicator (e.g. '-###.00') in your formatting string can change values and their data types.

**Usage Notes:**

| Required? | Data Type | Example Value |
|---|---|---|
| Yes | String (column reference), function, or Integer or Decimal literal | `MyPrice` |

### number_format_string

String value indicating the number format to apply to the input values.

> **NOTE:** You cannot create number format strings in which a `0` value appears before a `#` value. The following example strings are not supported: #.#0, #.#0#, #.#00

**Usage Notes:**

| Required? | Data Type | Example Value |
|---|---|---|
| Yes | String | `'###.00'` |

## grouping_separator

The string used to separate a group of digits. For example, a comma ( , ) is used as a grouping separator in the U.S.A ("10,000"), whereas space is used in France ("10 000").

> **NOTE:** If a space is used as the grouping separator, then any space values between a currency indicator and digits are automatically trimmed. A grouping separator should not be inserted between a currency indicator and a digit.

> **NOTE**: Using invalid separators or wrong separators may generate errors in your recipe step.

**Usage Notes:**

| Required? | Data Type | Example Value |
|---|---|---|
| No | String | `','` |

## decimal_separator

The string used to separate the integer part of a Decimal value from its fractional part. For example, a period( . ) is used as a decimal separator in the U.S.A ("1234.12"), whereas comma ( , ) is used in France ("1234,12").

**Usage Notes:**

| Required? | Data Type | Example Value |
|---|---|---|
| No | String | `'.'` |

## Examples

> **Tip:** For additional examples, see *Common Tasks*.

Designer Cloud Powered by Trifacta Enterprise Edition supports Java number formatting strings, with some exceptions.

## Example - formatting price and percentages

This example steps through how to manage number formatting for price and percentage data when you must perform some computations on the data in the application.

**Source:**

In this case, you need to compute sub-total and totals columns.

| OrderId | Qty | UnitPrice | Discount | TaxRate |
|---------|-----|-----------|----------|---------|
| 1001 | 5 | $25.00 | 0% | 8.25% |
| 1002 | 15 | $39.99 | 5% | 8.25% |
| 1003 | 2 | $99.99 | 15% | 8.25% |
| 1004 | 100 | $999.99 | 0% | 8.25% |

**Transformation:**

When this data is first imported into the Transformer page, you might notice the following:

- The data type for `OrderId` is an Integer, when it should be treated as String data.
- The `UnitPrice`, `Discount`, and `TaxRate` columns are typed as String data because of the unit characters in the values.

> **NOTE:** Where possible, remove currency and three-digit separators from your numeric data prior to import.

You can re-type the `OrderId` column to String without issue. If you retype the other three columns, all values are mismatched. You can use the following transforms to remove the currency and percentage notation. The first transform removes the trailing `%` sign from every value across all columns using a Trifacta pattern.

| | |
|---|---|
| **Transformation Name** | `Replace text or pattern` |
| **Parameter: Columns** | `All` |
| **Parameter: Find** | `` `\%{end}` `` |
| **Parameter: Replace with** | `''` |

You can use a similar one to remove the `$` sign at the beginning of values:

| | |
|---|---|
| **Transformation Name** | `Replace text or pattern` |
| **Parameter: Columns** | `All` |
| **Parameter: Find** | `` `{start}\$` `` |
| **Parameter: Replace with** | `''` |

When both are applied, you can see that the data types of each column is updated to a numeric type: Integer or Decimal. Now, you can perform the following computations:

| | |
|---|---|
| **Transformation Name** | `New formula` |
| **Parameter: Formula type** | `Single row formula` |

| Parameter: Formula | (Qty * UnitPrice) |
|---|---|
| Parameter: New column name | 'SubTotal' |

You can use the new `SubTotal` column as the basis for computing the `DiscountedTotal` column, which factors in discounts:

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | (SubTotal - (SubTotal * (Discount / 100))) |
| Parameter: New column name | 'DiscountedTotal' |

The `Total` column applies the tax to the `DiscountedTotal` column:

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | DiscountedTotal * (1 + (TaxRate / 100)) |
| Parameter: New column name | 'Total' |

Because of the math operations that have been applied to the original data, your values might no longer look like dollar information. You can now apply price formatting to your columns. The following changes the number format for the `SubTotal` column:

| Transformation Name | Edit column with formula |
|---|---|
| Parameter: Columns | SubTotal |
| Parameter: Formula | NUMFORMAT(SubTotal, '#.00', ',' ,'.') |

Note that the leading $ was not added back to the data, which changes the data type to String. You can apply this transform to the `Price`, `DiscountedTotal`, and `Total` columns.

> **NOTE:** The data types for your columns should match the expected inputs for your downstream analytics system.

The `Discount` and `TaxRate` values should be converted to Decimals. The following adjusts the `Discount` column:

| Transformation Name | Edit column with formula |
|---|---|
| Parameter: Columns | Discount |
| Parameter: Formula | (Discount / 100) |

**Results:**

The output data should look like the following:

| OrderId | Qty | UnitPrice | SubTotal | Discount | DiscountedTotal | TaxRate | Total |
|---------|-----|-----------|----------|----------|-----------------|---------|-------|
| 1001 | 5 | 25.00 | 125.00 | 0 | 125.00 | 0.0825 | 135.31 |
| 1002 | 15 | 39.99 | 599.85 | 0.05 | 569.86 | 0.0825 | 616.87 |
| 1003 | 2 | 99.99 | 199.98 | 0.15 | 169.98 | 0.0825 | 184.01 |
| 1004 | 100 | 999.99 | 99999.00 | 0 | 99999.00 | 0.0825 | 108248.92 |

# ADD Function

Returns the value of summing the first argument and the second argument. Equivalent to the + operator.

- Each argument can be a literal Integer or Decimal number, a function returning a number, or a reference to a column containing numeric values.

> **NOTE:** Within an expression, you might choose to use the corresponding operator, instead of this function. For more information, see *Numeric Operators*.

**Wrangle vs. SQL:** This function is part of Wrangle , a proprietary data transformation language. Wrangle is not SQL. For more information, see *Wrangle Language*.

## Basic Usage

```
add(2,3)
```

**Output:** Returns the sum of the values 2 and 3.

## Syntax and Arguments

```
add(value1, value2)
```

| Argument | Required? | Data Type | Description |
|----------|-----------|-----------|-------------|
| value1 | Y | string | The first value must be an Integer or Decimal literal, column reference, or expression that evaluates to one of those two numeric types. |
| value2 | Y | string | The first value must be an Integer or Decimal literal, column reference, or expression that evaluates to one of those two numeric types. |

For more information on syntax standards, see *Language Documentation Syntax Notes*.

**value1, value2**

Integer or Decimal expressions, column references or literals to sum together.

- Missing or mismatched values generate missing string results.

**Usage Notes:**

| Required? | Data Type | Example Value |
|-----------|-----------|---------------|
| Yes | Literal, function, or column reference returning an Integer or Decimal value | `myScore * 10` |

## Examples

> **Tip:** For additional examples, see *Common Tasks*.

### Example - Numeric Functions

This example demonstrates how to use numeric functions to perform computations in your recipe steps.

**Functions:**

| Item | Description |
|------|-------------|
| ADD Function | Returns the value of summing the first argument and the second argument. Equivalent to the + operator. |
| MOD Function | Returns the modulo value, which is the remainder of dividing the first argument by the second argument. Equivalent to the % operator. |
| NEGATE Function | Returns the opposite of the value that is the first argument. Equivalent to the − operator placed in front of the argument. |
| SUBTRACT Function | Returns the value of subtracting the second argument from the first argument. Equivalent to the − operator. |
| MULTIPLY Function | Returns the value of multiplying the first argument by the second argument. Equivalent to the * operator. |
| DIVIDE Function | Returns the value of dividing the first argument by the second argument. Equivalent to the / operator. |
| LCM Function | Returns the least common multiple shared by the first and second arguments. |

**Source:**

| ValueA | ValueB |
|--------|--------|
| 8 | 2 |
| 10 | 4 |
| 15 | 10 |
| 5 | 6 |

**Transformation:**

Execute the following transformation steps:

| Transformation Name | New formula |
|---------------------|-------------|
| **Parameter: Formula type** | Single row formula |
| **Parameter: Formula** | ADD(ValueA, ValueB) |
| **Parameter: New column name** | 'add' |

| Transformation Name | New formula |
|---------------------|-------------|
| **Parameter: Formula type** | Single row formula |
| **Parameter: Formula** | SUBTRACT(ValueA, ValueB) |
| **Parameter: New column name** | 'subtract' |

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | MULTIPLY(ValueA, ValueB) |
| Parameter: New column name | 'multiply' |

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | DIVIDE(ValueA, ValueB) |
| Parameter: New column name | 'divide' |

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | MOD(ValueA, ValueB) |
| Parameter: New column name | 'mod' |

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | NEGATE(ValueA) |
| Parameter: New column name | 'negativeA' |

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | LCM(ValueA, ValueB) |
| Parameter: New column name | 'lcm' |

**Results:**

With a bit of cleanup, your dataset results might look like the following:

| ValueA | ValueB | lcm | negativeA | mod | divide | multiply | subtract | add |
|---|---|---|---|---|---|---|---|---|
| 8 | 2 | 8 | -8 | 0 | 4 | 16 | 6 | 10 |
| 10 | 4 | 20 | -10 | 2 | 2.5 | 40 | 6 | 14 |
| 15 | 10 | 30 | -15 | 5 | 1.5 | 150 | 5 | 25 |
| 5 | 6 | 30 | -5 | 5 | 0.833333333 | 30 | -1 | 11 |

# SUBTRACT Function

Returns the value of subtracting the second argument from the first argument. Equivalent to the – operator.

- Each argument can be a literal Integer or Decimal number, a function returning a number, or a reference to a column containing numeric values.

> **NOTE:** Within an expression, you might choose to use the corresponding operator, instead of this function. For more information, see *Numeric Operators*.

**Wrangle vs. SQL:** This function is part of Wrangle , a proprietary data transformation language. Wrangle is not SQL. For more information, see *Wrangle Language*.

## Basic Usage

```
subtract(10,4)
```

**Output:** Returns the value 6.

## Syntax and Arguments

```
subtract(value1, value2)
```

| Argument | Required? | Data Type | Description |
|----------|-----------|-----------|-------------|
| value1 | Y | string | The first value must be an Integer or Decimal literal, column reference, or expression that evaluates to one of those two numeric types. |
| value2 | Y | string | The first value must be an Integer or Decimal literal, column reference, or expression that evaluates to one of those two numeric types. |

For more information on syntax standards, see *Language Documentation Syntax Notes*.

**value1, value2**

Integer or Decimal expressions, column references or literals to use in the subtraction.

- Missing or mismatched values generate missing string results.

**Usage Notes:**

| Required? | Data Type | Example Value |
|-----------|-----------|---------------|
| Yes | Literal, function, or column reference returning an Integer or Decimal value | `myScore * 10` |

## Examples

> **Tip:** For additional examples, see *Common Tasks*.

## Example - Numeric Functions

This example demonstrates how to use numeric functions to perform computations in your recipe steps.

### Functions:

| Item | Description |
|------|-------------|
| ADD Function | Returns the value of summing the first argument and the second argument. Equivalent to the + operator. |
| MOD Function | Returns the modulo value, which is the remainder of dividing the first argument by the second argument. Equivalent to the % operator. |
| NEGATE Function | Returns the opposite of the value that is the first argument. Equivalent to the − operator placed in front of the argument. |
| SUBTRACT Function | Returns the value of subtracting the second argument from the first argument. Equivalent to the − operator. |
| MULTIPLY Function | Returns the value of multiplying the first argument by the second argument. Equivalent to the * operator. |
| DIVIDE Function | Returns the value of dividing the first argument by the second argument. Equivalent to the / operator. |
| LCM Function | Returns the least common multiple shared by the first and second arguments. |

### Source:

| ValueA | ValueB |
|--------|--------|
| 8 | 2 |
| 10 | 4 |
| 15 | 10 |
| 5 | 6 |

### Transformation:

Execute the following transformation steps:

| Transformation Name | New formula |
|---------------------|-------------|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | ADD(ValueA, ValueB) |
| Parameter: New column name | 'add' |

| Transformation Name | New formula |
|---------------------|-------------|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | SUBTRACT(ValueA, ValueB) |
| Parameter: New column name | 'subtract' |

| Transformation Name | New formula |
|---------------------|-------------|
| Parameter: Formula type | Single row formula |

| Parameter: Formula | MULTIPLY(ValueA, ValueB) |
|---|---|
| Parameter: New column name | 'multiply' |

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | DIVIDE(ValueA, ValueB) |
| Parameter: New column name | 'divide' |

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | MOD(ValueA, ValueB) |
| Parameter: New column name | 'mod' |

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | NEGATE(ValueA) |
| Parameter: New column name | 'negativeA' |

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | LCM(ValueA, ValueB) |
| Parameter: New column name | 'lcm' |

**Results:**

With a bit of cleanup, your dataset results might look like the following:

| ValueA | ValueB | lcm | negativeA | mod | divide | multiply | subtract | add |
|---|---|---|---|---|---|---|---|---|
| 8 | 2 | 8 | -8 | 0 | 4 | 16 | 6 | 10 |
| 10 | 4 | 20 | -10 | 2 | 2.5 | 40 | 6 | 14 |
| 15 | 10 | 30 | -15 | 5 | 1.5 | 150 | 5 | 25 |
| 5 | 6 | 30 | -5 | 5 | 0.833333333 | 30 | -1 | 11 |

# MULTIPLY Function

Returns the value of multiplying the first argument by the second argument. Equivalent to the `*` operator.

- Each argument can be a literal Integer or Decimal number, a function returning a number, or a reference to a column containing numeric values.

> **NOTE:** Within an expression, you might choose to use the corresponding operator, instead of this function. For more information, see *Numeric Operators*.

**Wrangle vs. SQL:** This function is part of Wrangle , a proprietary data transformation language. Wrangle is not SQL. For more information, see *Wrangle Language*.

## Basic Usage

```
multiply(10,4)
```

**Output:** Returns the multiplication of `10` and `4`, which is `40`.

## Syntax and Arguments

```
multiply(value1, value2)
```

| Argument | Required? | Data Type | Description |
|----------|-----------|-----------|-------------|
| value1 | Y | string | The first value must be an Integer or Decimal literal, column reference, or expression that evaluates to one of those two numeric types. |
| value2 | Y | string | The second value must be an Integer or Decimal literal, column reference, or expression that evaluates to one of those two numeric types. |

For more information on syntax standards, see *Language Documentation Syntax Notes*.

**value1, value2**

Integer or Decimal expressions, column references or literals to multiply together.

- Missing or mismatched values generate missing string results.

**Usage Notes:**

| Required? | Data Type | Example Value |
|-----------|-----------|---------------|
| Yes | Literal, function, or column reference returning an Integer or Decimal value | 15 |

## Examples

> **Tip:** For additional examples, see *Common Tasks*.

### Example - Numeric Functions

This example demonstrates how to use numeric functions to perform computations in your recipe steps.

**Functions:**

| Item | Description |
|------|-------------|
| ADD Function | Returns the value of summing the first argument and the second argument. Equivalent to the + operator. |
| MOD Function | Returns the modulo value, which is the remainder of dividing the first argument by the second argument. Equivalent to the % operator. |
| NEGATE Function | Returns the opposite of the value that is the first argument. Equivalent to the − operator placed in front of the argument. |
| SUBTRACT Function | Returns the value of subtracting the second argument from the first argument. Equivalent to the − operator. |
| MULTIPLY Function | Returns the value of multiplying the first argument by the second argument. Equivalent to the * operator. |
| DIVIDE Function | Returns the value of dividing the first argument by the second argument. Equivalent to the / operator. |
| LCM Function | Returns the least common multiple shared by the first and second arguments. |

**Source:**

| ValueA | ValueB |
|--------|--------|
| 8 | 2 |
| 10 | 4 |
| 15 | 10 |
| 5 | 6 |

**Transformation:**

Execute the following transformation steps:

| Transformation Name | New formula |
|---------------------|-------------|
| **Parameter: Formula type** | Single row formula |
| **Parameter: Formula** | ADD(ValueA, ValueB) |
| **Parameter: New column name** | 'add' |

| Transformation Name | New formula |
|---------------------|-------------|
| **Parameter: Formula type** | Single row formula |
| **Parameter: Formula** | SUBTRACT(ValueA, ValueB) |
| **Parameter: New column name** | 'subtract' |

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | MULTIPLY(ValueA, ValueB) |
| Parameter: New column name | 'multiply' |

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | DIVIDE(ValueA, ValueB) |
| Parameter: New column name | 'divide' |

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | MOD(ValueA, ValueB) |
| Parameter: New column name | 'mod' |

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | NEGATE(ValueA) |
| Parameter: New column name | 'negativeA' |

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | LCM(ValueA, ValueB) |
| Parameter: New column name | 'lcm' |

**Results:**

With a bit of cleanup, your dataset results might look like the following:

| ValueA | ValueB | lcm | negativeA | mod | divide | multiply | subtract | add |
|---|---|---|---|---|---|---|---|---|
| 8 | 2 | 8 | -8 | 0 | 4 | 16 | 6 | 10 |
| 10 | 4 | 20 | -10 | 2 | 2.5 | 40 | 6 | 14 |
| 15 | 10 | 30 | -15 | 5 | 1.5 | 150 | 5 | 25 |
| 5 | 6 | 30 | -5 | 5 | 0.833333333 | 30 | -1 | 11 |

# DIVIDE Function

Returns the value of dividing the first argument by the second argument. Equivalent to the / operator.

- Each argument can be a literal Integer or Decimal number, a function returning a number, or a reference to a column containing numeric values.

> **NOTE:** Within an expression, you might choose to use the corresponding operator, instead of this function. For more information, see *Numeric Operators*.

**Wrangle vs. SQL:** This function is part of Wrangle , a proprietary data transformation language. Wrangle is not SQL. For more information, see *Wrangle Language*.

## Basic Usage

```
divide(20,4)
```

**Output:** The value 20 is divided by 4 and returned.

## Syntax and Arguments

```
divide(value1, value2)
```

| Argument | Required? | Data Type | Description |
|----------|-----------|-----------|-------------|
| value1 | Y | string | The first value must be an Integer or Decimal literal, column reference, or expression that evaluates to one of those two numeric types. |
| value2 | Y | string | The first value must be an Integer or Decimal literal, column reference, or expression that evaluates to one of those two numeric types. |

For more information on syntax standards, see *Language Documentation Syntax Notes*.

**value1, value2**

Integer or Decimal expressions, column references or literals.

- Missing or mismatched values generate missing string results.

**Usage Notes:**

| Required? | Data Type | Example Value |
|-----------|-----------|---------------|
| Yes | Literal, function, or column reference returning an Integer or Decimal value | 15 |

## Examples

> **Tip:** For additional examples, see *Common Tasks*.

**Example - Numeric Functions**

This example demonstrates how to use numeric functions to perform computations in your recipe steps.

**Functions:**

| Item | Description |
|---|---|
| ADD Function | Returns the value of summing the first argument and the second argument. Equivalent to the + operator. |
| MOD Function | Returns the modulo value, which is the remainder of dividing the first argument by the second argument. Equivalent to the % operator. |
| NEGATE Function | Returns the opposite of the value that is the first argument. Equivalent to the − operator placed in front of the argument. |
| SUBTRACT Function | Returns the value of subtracting the second argument from the first argument. Equivalent to the − operator. |
| MULTIPLY Function | Returns the value of multiplying the first argument by the second argument. Equivalent to the * operator. |
| DIVIDE Function | Returns the value of dividing the first argument by the second argument. Equivalent to the / operator. |
| LCM Function | Returns the least common multiple shared by the first and second arguments. |

**Source:**

| ValueA | ValueB |
|---|---|
| 8 | 2 |
| 10 | 4 |
| 15 | 10 |
| 5 | 6 |

**Transformation:**

Execute the following transformation steps:

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | ADD(ValueA, ValueB) |
| Parameter: New column name | 'add' |

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | SUBTRACT(ValueA, ValueB) |
| Parameter: New column name | 'subtract' |

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |

| Parameter: Formula | MULTIPLY(ValueA, ValueB) |
|---|---|
| Parameter: New column name | 'multiply' |

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | DIVIDE(ValueA, ValueB) |
| Parameter: New column name | 'divide' |

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | MOD(ValueA, ValueB) |
| Parameter: New column name | 'mod' |

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | NEGATE(ValueA) |
| Parameter: New column name | 'negativeA' |

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | LCM(ValueA, ValueB) |
| Parameter: New column name | 'lcm' |

**Results:**

With a bit of cleanup, your dataset results might look like the following:

| ValueA | ValueB | lcm | negativeA | mod | divide | multiply | subtract | add |
|---|---|---|---|---|---|---|---|---|
| 8 | 2 | 8 | -8 | 0 | 4 | 16 | 6 | 10 |
| 10 | 4 | 20 | -10 | 2 | 2.5 | 40 | 6 | 14 |
| 15 | 10 | 30 | -15 | 5 | 1.5 | 150 | 5 | 25 |
| 5 | 6 | 30 | -5 | 5 | 0.833333333 | 30 | -1 | 11 |

# MOD Function

Returns the modulo value, which is the remainder of dividing the first argument by the second argument. Equivalent to the `%` operator.

- Each argument can be a literal Integer or Decimal number, a function returning a number, or a reference to a column containing numeric values.

> **NOTE:** Within an expression, you might choose to use the corresponding operator, instead of this function. For more information, see *Numeric Operators*.

**Wrangle vs. SQL:** This function is part of Wrangle , a proprietary data transformation language. Wrangle is not SQL. For more information, see *Wrangle Language*.

## Basic Usage

```
mod(14,3)
```

**Output:** Returns remainder of the value `14` divided by `3`, which is `2`.

## Syntax and Arguments

```
mod(value1, value2)
```

| Argument | Required? | Data Type | Description |
|----------|-----------|-----------|-------------|
| value1 | Y | string | The first value must be an Integer or Decimal literal, column reference, or expression that evaluates to one of those two numeric types. |
| value2 | Y | string | The first value must be an Integer or Decimal literal, column reference, or expression that evaluates to one of those two numeric types. |

For more information on syntax standards, see *Language Documentation Syntax Notes*.

**value1, value2**

Integer or Decimal expressions, column references or literals.

- Missing or mismatched values generate missing string results.

**Usage Notes:**

| Required? | Data Type | Example Value |
|-----------|-----------|---------------|
| Yes | Literal, function, or column reference returning an Integer or Decimal value | 15 |

## Examples

> **Tip:** For additional examples, see *Common Tasks*.

### Example - Numeric Functions

This example demonstrates how to use numeric functions to perform computations in your recipe steps.

**Functions:**

| Item | Description |
|---|---|
| ADD Function | Returns the value of summing the first argument and the second argument. Equivalent to the + operator. |
| MOD Function | Returns the modulo value, which is the remainder of dividing the first argument by the second argument. Equivalent to the % operator. |
| NEGATE Function | Returns the opposite of the value that is the first argument. Equivalent to the − operator placed in front of the argument. |
| SUBTRACT Function | Returns the value of subtracting the second argument from the first argument. Equivalent to the − operator. |
| MULTIPLY Function | Returns the value of multiplying the first argument by the second argument. Equivalent to the * operator. |
| DIVIDE Function | Returns the value of dividing the first argument by the second argument. Equivalent to the / operator. |
| LCM Function | Returns the least common multiple shared by the first and second arguments. |

**Source:**

| ValueA | ValueB |
|---|---|
| 8 | 2 |
| 10 | 4 |
| 15 | 10 |
| 5 | 6 |

**Transformation:**

Execute the following transformation steps:

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | ADD(ValueA, ValueB) |
| Parameter: New column name | 'add' |

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | SUBTRACT(ValueA, ValueB) |
| Parameter: New column name | 'subtract' |

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |

| Parameter: Formula | MULTIPLY(ValueA, ValueB) |
|---|---|
| Parameter: New column name | 'multiply' |

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | DIVIDE(ValueA, ValueB) |
| Parameter: New column name | 'divide' |

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | MOD(ValueA, ValueB) |
| Parameter: New column name | 'mod' |

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | NEGATE(ValueA) |
| Parameter: New column name | 'negativeA' |

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | LCM(ValueA, ValueB) |
| Parameter: New column name | 'lcm' |

**Results:**

With a bit of cleanup, your dataset results might look like the following:

| ValueA | ValueB | lcm | negativeA | mod | divide | multiply | subtract | add |
|---|---|---|---|---|---|---|---|---|
| 8 | 2 | 8 | -8 | 0 | 4 | 16 | 6 | 10 |
| 10 | 4 | 20 | -10 | 2 | 2.5 | 40 | 6 | 14 |
| 15 | 10 | 30 | -15 | 5 | 1.5 | 150 | 5 | 25 |
| 5 | 6 | 30 | -5 | 5 | 0.833333333 | 30 | -1 | 11 |

# NEGATE Function

Returns the opposite of the value that is the first argument. Equivalent to the – operator placed in front of the argument.

- The argument can be a literal Integer or Decimal number, a function returning a number, or a reference to a column containing numeric values.

> **NOTE:** Within an expression, you might choose to use the corresponding operator, instead of this function. For more information, see *Numeric Operators*.

**Wrangle vs. SQL:** This function is part of Wrangle , a proprietary data transformation language. Wrangle is not SQL. For more information, see *Wrangle Language*.

## Basic Usage

```
negate(MyValue)
```

**Output:** Returns the opposite of the value in the `MyValue` column.

## Syntax and Arguments

```
negate(value1)
```

| Argument | Required? | Data Type | Description |
|----------|-----------|-----------|-------------|
| value1 | Y | string | The first value must be an Integer or Decimal literal, column reference, or expression that evaluates to one of those two numeric types. |

For more information on syntax standards, see *Language Documentation Syntax Notes*.

**value1**

Integer or Decimal expressions, column references or literals.

- Missing or mismatched values generate missing string results.

**Usage Notes:**

| Required? | Data Type | Example Value |
|-----------|-----------|---------------|
| Yes | Literal, function, or column reference returning an Integer or Decimal value | 15 |

## Examples

> **Tip:** For additional examples, see *Common Tasks*.

## Example - Numeric Functions

This example demonstrates how to use numeric functions to perform computations in your recipe steps.

### Functions:

| Item | Description |
|------|-------------|
| ADD Function | Returns the value of summing the first argument and the second argument. Equivalent to the + operator. |
| MOD Function | Returns the modulo value, which is the remainder of dividing the first argument by the second argument. Equivalent to the % operator. |
| NEGATE Function | Returns the opposite of the value that is the first argument. Equivalent to the − operator placed in front of the argument. |
| SUBTRACT Function | Returns the value of subtracting the second argument from the first argument. Equivalent to the − operator. |
| MULTIPLY Function | Returns the value of multiplying the first argument by the second argument. Equivalent to the * operator. |
| DIVIDE Function | Returns the value of dividing the first argument by the second argument. Equivalent to the / operator. |
| LCM Function | Returns the least common multiple shared by the first and second arguments. |

### Source:

| ValueA | ValueB |
|--------|--------|
| 8 | 2 |
| 10 | 4 |
| 15 | 10 |
| 5 | 6 |

### Transformation:

Execute the following transformation steps:

| Transformation Name | New formula |
|---------------------|-------------|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | ADD(ValueA, ValueB) |
| Parameter: New column name | 'add' |

| Transformation Name | New formula |
|---------------------|-------------|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | SUBTRACT(ValueA, ValueB) |
| Parameter: New column name | 'subtract' |

| Transformation Name | New formula |
|---------------------|-------------|

| Parameter: Formula type | Single row formula |
|---|---|
| Parameter: Formula | MULTIPLY(ValueA, ValueB) |
| Parameter: New column name | 'multiply' |

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | DIVIDE(ValueA, ValueB) |
| Parameter: New column name | 'divide' |

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | MOD(ValueA, ValueB) |
| Parameter: New column name | 'mod' |

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | NEGATE(ValueA) |
| Parameter: New column name | 'negativeA' |

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | LCM(ValueA, ValueB) |
| Parameter: New column name | 'lcm' |

**Results:**

With a bit of cleanup, your dataset results might look like the following:

| ValueA | ValueB | lcm | negativeA | mod | divide | multiply | subtract | add |
|---|---|---|---|---|---|---|---|---|
| 8 | 2 | 8 | -8 | 0 | 4 | 16 | 6 | 10 |
| 10 | 4 | 20 | -10 | 2 | 2.5 | 40 | 6 | 14 |
| 15 | 10 | 30 | -15 | 5 | 1.5 | 150 | 5 | 25 |
| 5 | 6 | 30 | -5 | 5 | 0.833333333 | 30 | -1 | 11 |

# SIGN Function

Computes the positive or negative sign of a given numeric value. The value can be a Decimal or Integer literal, a function returning Decimal or Integer, or a reference to a column containing numeric values.

- For positive values, this function returns `1`.
- For negative values, this function returns `-1`.
- For the value `0`, this function returns `0`.

**Wrangle vs. SQL:** This function is part of Wrangle , a proprietary data transformation language. Wrangle is not SQL. For more information, see *Wrangle Language*.

## Basic Usage

**Column reference example:**

```
sign(MyInteger)
```

**Output:** Returns the sign of the value found in the `MyInteger` column.

**Numeric literal example:**

```
(sign(MyInteger) == -1)
```

**Output:** Returns `true` if the sign of the entry in the `MyInteger` column is `-1`.

## Syntax and Arguments

```
sign(numeric_value)
```

| Argument | Required? | Data Type | Description |
|---|---|---|---|
| numeric_value | Y | Decimal or Integer | Decimal or Integer literal, function returning Decimal or Integer, or name of column to apply to the function |

For more information on syntax standards, see *Language Documentation Syntax Notes*.

**numeric_value**

Numeric literal, function returning numeric literal, or name of the column containing values the sign of which are to be computed.

- Missing input values generate missing results.
- Literal numeric values should not be quoted. Quoted values are treated as strings.
- Multiple columns and wildcards are not supported.

**Usage Notes:**

| Required? | Data Type | Example Value |
|---|---|---|
| Yes | String (column reference) or Integer or Decimal value | -10.5 |

# Examples

> **Tip:** For additional examples, see *Common Tasks*.

**Example - Basic SIGN function**

**Source:**

Your source data looks like the following, which measures coordinate distances from a fixed point on a grid:

| X | Y |
|------|-------|
| -2 | 4 |
| -6.2 | -2 |
| 0 | -4.2 |
| 4 | 4 |
| 15 | -0.05 |

**Transformation:**

You can use the following transform to derive the sign values of these columns:

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | sign(X) |
| Parameter: New column name | 'signX' |

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | sign(Y) |
| Parameter: New column name | 'signY' |

Using these two columns, you can assign each set of coordinates into a quadrant. For ease of reading, the following has been broken into two separate tranformations:

| Transformation Name | Conditional column |
|---|---|
| Parameter: Condition type | Case on custom conditions |
| Parameter: Case 01 - Condition | (signX == 1) && (signY == -1) |
| Parameter: Case 01 - Value | 'lower-right' |
| Parameter: Case 02 - Condition | (signX == 1) && (signY == 1) |

| | |
|---|---|
| **Parameter: Case 02 - Value** | `'upper-right'` |
| **Parameter: Default value** | `'line'` |
| **Parameter: New column name** | `'q1'` |

| | |
|---|---|
| **Transformation Name** | `Conditional column` |
| **Parameter: Condition type** | `Case on custom conditions` |
| **Parameter: Case 01 - Condition** | `(signX == -1) && (signY == -1)` |
| **Parameter: Case 01 - Value** | `'lower-left'` |
| **Parameter: Case 02 - Condition** | `(signX == -1) && (signY == 1)` |
| **Parameter: Case 02 - Value** | `'upper-left'` |
| **Parameter: Default value** | `'line'` |
| **Parameter: New column name** | `'q2'` |

Then, you can merge the two columns together:

| | |
|---|---|
| **Transformation Name** | `Merge columns` |
| **Parameter: Columns** | `q1,q2` |
| **Parameter: Separator** | `''` |
| **Parameter: New column name** | `'quadrant'` |

**Results:**

| X | Y | signX | signY | quadrant |
|---|---|---|---|---|
| -2 | 4 | -1 | 1 | upper-left |
| -6.2 | -2 | -1 | -1 | lower-left |
| 0 | -4.2 | 0 | -1 | line |
| 4 | 4 | 1 | 1 | upper-right |
| 15 | -0.05 | 1 | -1 | lower-right |

# LCM Function

Returns the least common multiple shared by the first and second arguments.

- Each argument can be a literal Integer number, a function returning an Integer, or a reference to a column containing Integer values.

**Wrangle vs. SQL:** This function is part of Wrangle , a proprietary data transformation language. Wrangle is not SQL. For more information, see *Wrangle Language*.

## Basic Usage

```
lcm(10,4)
```

**Output:** Returns the least common multiple between values `10` and `4`, which is `20`.

## Syntax and Arguments

```
lcm(value1, value2)
```

| Argument | Required? | Data Type | Description |
|---|---|---|---|
| value1 | Y | string | The first value must be an Integer literal, column reference, or expression that evaluates to an Integer value. |
| value2 | Y | string | The first value must be an Integer literal, column reference, or expression that evaluates to an Integer value. |

For more information on syntax standards, see *Language Documentation Syntax Notes*.

**value1, value2**

Integer expressions, column references or literals to multiply together.

- Missing or mismatched values generate missing string results.

**Usage Notes:**

| Required? | Data Type | Example Value |
|---|---|---|
| Yes | Literal, function, or column reference returning an Integer value | 15 |

## Examples

> **Tip:** For additional examples, see *Common Tasks*.

**Example - Basic LCM function**

**Source:**

| string | repeat_count |
|--------|--------------|
| ha | 0 |
| ha | 1 |
| ha | 1.5 |
| ha | 2 |
| ha | -2 |

## Transformation:

| | |
|--|--|
| **Transformation Name** | New formula |
| **Parameter: Formula type** | Single row formula |
| **Parameter: Formula** | repeat(string, repeat_count) |
| **Parameter: New column name** | 'repeat_string' |

## Results:

| string | repeat_count | repeat_string |
|--------|--------------|---------------|
| ha | 0 | |
| ha | 1 | ha |
| ha | 1.5 | |
| ha | 2 | haha |
| ha | -2 | |

# ABS Function

Computes the absolute value of a given numeric value. The value can be a Decimal or Integer literal or a reference to a column containing numeric values.

**Wrangle vs. SQL:** This function is part of Wrangle , a proprietary data transformation language. Wrangle is not SQL. For more information, see *Wrangle Language*.

## Basic Usage

**Column reference example:**

```
<span>abs(MyInteger)</span>
```

**Output:** Returns the absolute value of each value found in the `MyInteger` column.

**Numeric literal example:**

```
<span>(</span><span>abs(</span><span>MyInteger</span><span>) </span><span>== 5)</span>
```

**Output:** Returns `true` if the absolute value of the entry in the `MyInteger` column is 5.

## Syntax and Arguments

```
<span>abs(numeric_value)</span>
```

| Argument | Required? | Data Type | Description |
|---|---|---|---|
| numeric_value | Y | string, decimal, or integer | Name of column or Decimal or Integer literal to apply to the function |

For more information on syntax standards, see *Language Documentation Syntax Notes*.

**numeric_value**

Name of the column or numeric literal whose absolute value is to be computed.

- Missing input values generate missing results.
- Literal numeric values should not be quoted. Quoted values are treated as strings.
- Multiple columns and wildcards are not supported.

**Usage Notes:**

| Required? | Data Type | Example Value |
|---|---|---|
| Yes | String (column reference) or Integer or Decimal value | -10.5 |

## Examples

> **Tip:** For additional examples, see *Common Tasks*.

**Example - Basic ABS function**

**Source:**

Your source data looks like the following, which measures coordinate distances from a fixed point on a grid:

| X | Y |
|------|-------|
| -2 | 4 |
| -6.2 | -2 |
| 0 | -4.2 |
| 4 | 4 |
| 15 | -0.05 |

**Transform:**

You can use the following transform to derive the absolute values of these columns, which now measure distance from the fixed point:

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | abs(X) |
| Parameter: New column name | 'distanceX' |

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | abs(y) |
| Parameter: New column name | 'distanceY' |

**Results:**

| X | Y | distanceX | distanceY |
|------|-------|-----------|-----------|
| -2 | 4 | 2 | 4 |
| -6.2 | -2 | 6.2 | 2 |
| 0 | -4.2 | 0 | 4.2 |
| 4 | 4 | 4 | 4 |
| 15 | -0.05 | 15 | 0.05 |

You can then use POW and SQRT functions to compute the total distance.

# EXP Function

Computes the value of *e* raised to the specified power. The value can be a Decimal or Integer literal or a reference to a column containing numeric values.

- *e* or Euler's Number is the value that the following equation approaches as *n* grows large: $(1 + (1/n))^n$.

**Wrangle vs. SQL:** This function is part of Wrangle , a proprietary data transformation language. Wrangle is not SQL. For more information, see *Wrangle Language*.

## Basic Usage

**Numeric literal example:**

```
exp(1)
```

**Output:** Returns the value of $e^1$, which is approximately 2.718281828.

**Column reference example:**

```
exp(MyValue)
```

**Output:** Returns the value of *e* to the power of the value in the `MyValue` column.

## Syntax and Arguments

```
exp(numeric_value)
```

| Argument | Required? | Data Type | Description |
|---|---|---|---|
| numeric_value | Y | string, decimal, or integer | Name of column or Decimal or Integer literal to apply to the function |

For more information on syntax standards, see *Language Documentation Syntax Notes*.

**numeric_value**

Name of the column or numeric literal whose value is the exponent of *e*.

- Missing input values generate missing results.
- Literal numeric values should not be quoted. Quoted values are treated as strings.
- Multiple columns and wildcards are not supported.

**Usage Notes:**

| Required? | Data Type | Example Value |
|---|---|---|
| Yes | String (column reference) or Integer or Decimal literal | 2.3 |

# Examples

> **Tip:** For additional examples, see *Common Tasks*.

### Example - Exponential functions

This example demonstrates the exponential functions.

**Functions**:

| Item | Description |
|---|---|
| EXP Function | Computes the value of *e* raised to the specified power. The value can be a Decimal or Integer literal or a reference to a column containing numeric values. |
| LN Function | Computes the natural logarithm of an input value. The value can be a Decimal or Integer literal or a reference to a column containing numeric values. |
| LOG Function | Computes the logarithm of the first argument with a base of the second argument. |
| POW Function | Computes the value of the first argument raised to the value of the second argument. |

**Source:**

| rowNum | X |
|---|---|
| 1 | -2 |
| 2 | 1 |
| 3 | 0 |
| 4 | 1 |
| 5 | 2 |
| 6 | 3 |
| 7 | 4 |
| 8 | 5 |

**Transformation:**

| Transformation Name | New formula |
|---|---|
| **Parameter: Formula type** | Single row formula |
| **Parameter: Formula** | EXP (X) |
| **Parameter: New column name** | 'expX' |

| Transformation Name | New formula |
|---|---|

| Parameter: Formula type | Single row formula |
|---|---|
| Parameter: Formula | LN (expX) |
| Parameter: New column name | 'ln_expX' |

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | LOG (X) |
| Parameter: New column name | 'logX' |

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | POW (10,logX) |
| Parameter: New column name | 'pow_logX' |

**Results:**

In the following, (null value) indicates that a null value is generated for the computation.

| rowNum | X | expX | ln_expX | logX | pow_logX |
|---|---|---|---|---|---|
| 1 | -2 | 0.1353352832366127 | -2 | (null value) | (null value) |
| 2 | -1 | 0.1353352832366127 | -0.9999999999999998 | (null value) | (null value) |
| 3 | 0 | 1 | 0 | (null value) | 0 |
| 4 | 1 | 2.718281828459045 | 1 | 0 | 1 |
| 5 | 2 | 7.3890560989306495 | 2 | 0.30102999566398114 | 1.9999999999999998 |
| 6 | 3 | 20.085536923187668 | 3 | 0.47712125471966244 | 3 |
| 7 | 4 | 54.59815003314423 | 4 | 0.6020599913279623 | 3.999999999999999 |
| 8 | 5 | 148.41315910257657 | 5 | 0.6989700043360187 | 4.999999999999999 |

# LOG Function

**Contents:**

- *Basic Usage*
- *Syntax and Arguments*
  - *result_numeric_value*
  - *base_numeric_value*
- *Examples*
  - *Example - Exponential functions*

---

Computes the logarithm of the first argument with a base of the second argument.

- First argument can be a Decimal or Integer literal or a reference to a column containing numeric values.
- Second argument, the base, must be an Integer value or column reference.

**Wrangle vs. SQL:** This function is part of Wrangle , a proprietary data transformation language. Wrangle is not SQL. For more information, see *Wrangle Language*.

## Basic Usage

**Numeric literal example:**

```
log(49, 7)
```

**Output:** Returns 2.

**Column reference example:**

```
log(MyValue, 5)
```

**Output:** Returns the exponent that raises 5 to yield the MyValue column.

## Syntax and Arguments

```
log(result_numeric_value, base_numeric_value)
```

| Argument | Required? | Data Type | Description |
|---|---|---|---|
| result_numeric_value | Y | string, decimal, or integer | Name of column or Decimal or Integer literal that is generated by the LOG function |
| base_numeric_value | Y | string, decimal, or integer | Name of column or Decimal or Integer literal that serves as the base for computing the LOG function |

For more information on syntax standards, see *Language Documentation Syntax Notes*.

**result_numeric_value**

Name of the column or numeric literal. Value must be greater than 0.

- Missing input values generate missing results.

- Literal numeric values should not be quoted. Quoted values are treated as strings.
- Multiple columns and wildcards are not supported.

**Usage Notes:**

| Required? | Data Type | Example Value |
|---|---|---|
| Yes | String (column reference) or Integer or Decimal literal | 49 |

### base_numeric_value

Name of the column or Integer literal that is used for the exponential calculation.

> **NOTE:** This base value must be a positive integer. If this value is not specified, 10 is used as the base value.

- Missing input values generate missing results.
- Literal numeric values should not be quoted. Quoted values are treated as strings.
- Multiple columns and wildcards are not supported.

**Usage Notes:**

| Required? | Data Type | Example Value |
|---|---|---|
| No | String (column reference) or Integer or Decimal literal | 7 |

## Examples

> **Tip:** For additional examples, see *Common Tasks*.

### Example - Exponential functions

This example demonstrates the exponential functions.

**Functions**:

| Item | Description |
|---|---|
| EXP Function | Computes the value of *e* raised to the specified power. The value can be a Decimal or Integer literal or a reference to a column containing numeric values. |
| LN Function | Computes the natural logarithm of an input value. The value can be a Decimal or Integer literal or a reference to a column containing numeric values. |
| LOG Function | Computes the logarithm of the first argument with a base of the second argument. |
| POW Function | Computes the value of the first argument raised to the value of the second argument. |

**Source:**

| rowNum | X |
|---|---|

| | |
|---|---|
| 1 | -2 |
| 2 | 1 |
| 3 | 0 |
| 4 | 1 |
| 5 | 2 |
| 6 | 3 |
| 7 | 4 |
| 8 | 5 |

**Transformation:**

| | |
|---|---|
| **Transformation Name** | New formula |
| **Parameter: Formula type** | Single row formula |
| **Parameter: Formula** | EXP (X) |
| **Parameter: New column name** | 'expX' |

| | |
|---|---|
| **Transformation Name** | New formula |
| **Parameter: Formula type** | Single row formula |
| **Parameter: Formula** | LN (expX) |
| **Parameter: New column name** | 'ln_expX' |

| | |
|---|---|
| **Transformation Name** | New formula |
| **Parameter: Formula type** | Single row formula |
| **Parameter: Formula** | LOG (X) |
| **Parameter: New column name** | 'logX' |

| | |
|---|---|
| **Transformation Name** | New formula |
| **Parameter: Formula type** | Single row formula |
| **Parameter: Formula** | POW (10,logX) |
| **Parameter: New column name** | 'pow_logX' |

**Results:**

In the following, `(null value)` indicates that a null value is generated for the computation.

| rowNum | X | expX | ln_expX | logX | pow_logX |
|---|---|---|---|---|---|
| 1 | -2 | 0.1353352832366127 | -2 | (null value) | (null value) |

| 2 | -1 | 0.1353352832366127 | -0.9999999999999998 | (null value) | (null value) |
| 3 | 0 | 1 | 0 | (null value) | 0 |
| 4 | 1 | 2.718281828459045 | 1 | 0 | 1 |
| 5 | 2 | 7.3890560989306495 | 2 | 0.30102999566398114 | 1.9999999999999998 |
| 6 | 3 | 20.085536923187668 | 3 | 0.47712125471966244 | 3 |
| 7 | 4 | 54.59815003314423 | 4 | 0.6020599913279623 | 3.999999999999999 |
| 8 | 5 | 148.41315910257657 | 5 | 0.6989700043360187 | 4.999999999999999 |

# LN Function

Computes the natural logarithm of an input value. The value can be a Decimal or Integer literal or a reference to a column containing numeric values.

- The **natural logarithm** of a value is the value of $e^x$ such that is the input value.

**Wrangle vs. SQL:** This function is part of Wrangle , a proprietary data transformation language. Wrangle is not SQL. For more information, see *Wrangle Language*.

## Basic Usage

**Numeric literal example:**

```
ln(10)
```

**Output:** Returns the value X, such that $e^x$, is 10. This value is approximately `2.302585092994046`.

**Column reference example:**

```
ln(MyValue)
```

**Output:** Returns the power to which *e* is raised to yield the value in the `MyValue` column.

## Syntax and Arguments

```
ln(numeric_value)
```

| Argument | Required? | Data Type | Description |
|---|---|---|---|
| numeric_value | Y | string, decimal, or integer | Name of column or Decimal or Integer literal to apply to the function |

For more information on syntax standards, see *Language Documentation Syntax Notes*.

**numeric_value**

Name of the column or numeric literal, the natural logarithm of which is to be computed.

- Missing input values generate missing results.
- Literal numeric values should not be quoted. Quoted values are treated as strings.
- Multiple columns and wildcards are not supported.

**Usage Notes:**

| Required? | Data Type | Example Value |
|---|---|---|
| Yes | String (column reference) or Integer or Decimal literal | 10 |

## Examples

> **Tip:** For additional examples, see *Common Tasks*.

### Example - Exponential Functions

This example demonstrates the exponential functions.

**Functions**:

| Item | Description |
|---|---|
| EXP Function | Computes the value of *e* raised to the specified power. The value can be a Decimal or Integer literal or a reference to a column containing numeric values. |
| LN Function | Computes the natural logarithm of an input value. The value can be a Decimal or Integer literal or a reference to a column containing numeric values. |
| LOG Function | Computes the logarithm of the first argument with a base of the second argument. |
| POW Function | Computes the value of the first argument raised to the value of the second argument. |

**Source:**

| rowNum | X |
|---|---|
| 1 | -2 |
| 2 | 1 |
| 3 | 0 |
| 4 | 1 |
| 5 | 2 |
| 6 | 3 |
| 7 | 4 |
| 8 | 5 |

**Transformation:**

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | EXP (X) |
| Parameter: New column name | 'expX' |

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | LN (expX) |

| | |
|---|---|
| **Parameter: New column name** | `'ln_expX'` |

| | |
|---|---|
| **Transformation Name** | `New formula` |
| **Parameter: Formula type** | `Single row formula` |
| **Parameter: Formula** | `LOG (X)` |
| **Parameter: New column name** | `'logX'` |

| | |
|---|---|
| **Transformation Name** | `New formula` |
| **Parameter: Formula type** | `Single row formula` |
| **Parameter: Formula** | `POW (10,logX)` |
| **Parameter: New column name** | `'pow_logX'` |

**Results:**

In the following, `(null value)` indicates that a null value is generated for the computation.

| rowNum | X | expX | ln_expX | logX | pow_logX |
|---|---|---|---|---|---|
| 1 | -2 | 0.1353352832366127 | -2 | (null value) | (null value) |
| 2 | -1 | 0.1353352832366127 | -0.9999999999999998 | (null value) | (null value) |
| 3 | 0 | 1 | 0 | (null value) | 0 |
| 4 | 1 | 2.718281828459045 | 1 | 0 | 1 |
| 5 | 2 | 7.3890560989306495 | 2 | 0.30102999566398114 | 1.9999999999999998 |
| 6 | 3 | 20.085536923187668 | 3 | 0.47712125471966244 | 3 |
| 7 | 4 | 54.59815003314423 | 4 | 0.6020599913279623 | 3.999999999999999 |
| 8 | 5 | 148.41315910257657 | 5 | 0.6989700043360187 | 4.999999999999999 |

# POW Function

**Contents:**

---

Computes the value of the first argument raised to the value of the second argument.

Each argument can be a Decimal or Integer literal or a reference to a column containing numeric values.

**Wrangle vs. SQL:** This function is part of Wrangle , a proprietary data transformation language. Wrangle is not SQL. For more information, see *Wrangle Language*.

## Basic Usage

**Numeric literal example:**

```
pow(10,3)
```

**Output:** Returns the value of $10^3$, which is `1000`.

**Column reference example:**

```
pow(MyValue,2)
```

**Output:** Returns the value of the `MyValue` column raised to the power of 2 (squared).

## Syntax and Arguments

```
pow(base_numeric_value, exp_numeric_value)
```

| Argument | Required? | Data Type | Description |
|---|---|---|---|
| base_numeric_value | Y | string, decimal, or integer | Name of column or Decimal or Integer literal that is the base value to be raised to the power of the second argument |
| exp_numeric_value | Y | string, decimal, or integer | Name of column or Decimal or Integer literal that is the power to which to raise the base value |

For more information on syntax standards, see *Language Documentation Syntax Notes*.

**base_numeric_value**

Name of the column or numeric literal whose values are used as the bases for the exponential computation.

- Missing input values generate missing results.

- Literal numeric values should not be quoted.
- Multiple columns and wildcards are not supported.

**Usage Notes:**

| Required? | Data Type | Example Value |
|-----------|-----------|---------------|
| Yes | String (column reference) or Integer or Decimal literal | 2.3 |

### exp_numeric_value

Name of the column or numeric literal whose values are used as the power to which the base-numeric value is raised.

- Missing input values generate missing results.
- Literal numeric values should not be quoted.
- Multiple columns and wildcards are not supported.

**Usage Notes:**

| Required? | Data Type | Example Value |
|-----------|-----------|---------------|
| Yes | String (column reference) or Integer or Decimal literal | 5 |

## Examples

> **Tip:** For additional examples, see *Common Tasks*.

### Example - Exponential functions

This example demonstrates the exponential functions.

**Functions**:

| Item | Description |
|------|-------------|
| EXP Function | Computes the value of *e* raised to the specified power. The value can be a Decimal or Integer literal or a reference to a column containing numeric values. |
| LN Function | Computes the natural logarithm of an input value. The value can be a Decimal or Integer literal or a reference to a column containing numeric values. |
| LOG Function | Computes the logarithm of the first argument with a base of the second argument. |
| POW Function | Computes the value of the first argument raised to the value of the second argument. |

**Source:**

| rowNum | X |
|--------|-----|
| 1 | -2 |
| 2 | 1 |

| 3 | 0 |
|---|---|
| 4 | 1 |
| 5 | 2 |
| 6 | 3 |
| 7 | 4 |
| 8 | 5 |

**Transformation:**

| Transformation Name | New formula |
|---|---|
| **Parameter: Formula type** | Single row formula |
| **Parameter: Formula** | EXP (X) |
| **Parameter: New column name** | 'expX' |

| Transformation Name | New formula |
|---|---|
| **Parameter: Formula type** | Single row formula |
| **Parameter: Formula** | LN (expX) |
| **Parameter: New column name** | 'ln_expX' |

| Transformation Name | New formula |
|---|---|
| **Parameter: Formula type** | Single row formula |
| **Parameter: Formula** | LOG (X) |
| **Parameter: New column name** | 'logX' |

| Transformation Name | New formula |
|---|---|
| **Parameter: Formula type** | Single row formula |
| **Parameter: Formula** | POW (10,logX) |
| **Parameter: New column name** | 'pow_logX' |

**Results:**

In the following, (null value) indicates that a null value is generated for the computation.

| rowNum | X | expX | ln_expX | logX | pow_logX |
|---|---|---|---|---|---|
| 1 | -2 | 0.1353352832366127 | -2 | (null value) | (null value) |
| 2 | -1 | 0.1353352832366127 | -0.9999999999999998 | (null value) | (null value) |
| 3 | 0 | 1 | 0 | (null value) | 0 |

| 4 | 1 | 2.718281828459045 | 1 | 0 | 1 |
| 5 | 2 | 7.3890560989306495 | 2 | 0.30102999566398114 | 1.9999999999999998 |
| 6 | 3 | 20.085536923187668 | 3 | 0.47712125471966244 | 3 |
| 7 | 4 | 54.59815003314423 | 4 | 0.6020599913279623 | 3.999999999999999 |
| 8 | 5 | 148.41315910257657 | 5 | 0.6989700043360187 | 4.999999999999999 |

### Example - Pythagorean Theorem

In this example, you learn how to compute exponentials and square roots on your numeric data.

**Functions**:

| Item | Description |
| --- | --- |
| POW Function | Computes the value of the first argument raised to the value of the second argument. |
| SQRT Function | Computes the square root of the input parameter. Input value can be a Decimal or Integer literal or a reference to a column containing numeric values. All generated values are non-negative. |

**Source:**

The dataset below contains values for x and y:

| X | Y |
| --- | --- |
| 3 | 4 |
| 4 | 9 |
| 8 | 10 |
| 30 | 40 |

**Transformation:**

You can use the following transformation to generate values for $z^2$.

> **NOTE:** Do not add this step to your recipe right now.

| Transformation Name | New formula |
| --- | --- |
| Parameter: Formula type | Single row formula |
| Parameter: Formula | (POW(x,2) + POW(y,2)) |
| Parameter: New column name | 'Z' |

You can see how column Z is generated as the sum of squares of the other two columns, which yields $z^2$.

Now, edit the transformation to wrap the value computation in a SQRT function. This step is done to compute the value for $z$, which is the distance between the two points based on the Pythagorean theorem.

| | |
|---|---|
| **Transformation Name** | New formula |
| **Parameter: Formula type** | Single row formula |
| **Parameter: Formula** | SQRT((POW(x,2) + POW(y,2))) |
| **Parameter: New column name** | 'Z' |

**Results:**

| X | Y | Z |
|---|---|---|
| 3 | 4 | 5 |
| 4 | 9 | 9.848857801796104 |
| 8 | 10 | 12.806248474865697 |
| 30 | 40 | 50 |

# SQRT Function

Computes the square root of the input parameter.  Input value can be a Decimal or Integer literal or a reference to a column containing numeric values. All generated values are non-negative.

**Wrangle vs. SQL:** This function is part of Wrangle , a proprietary data transformation language. Wrangle is not SQL. For more information, see *Wrangle Language*.

## Basic Usage

**Numeric literal example:**

```
sqrt(25)
```

**Output:** Returns the square root of 25, which is `5`.

**Column reference example:**

```
sqrt(MyValue)
```

**Output:** Returns the square root of the values of the `MyValue` column.

## Syntax and Arguments

```
sqrt(numeric_value)
```

| Argument | Required? | Data Type | Description |
|---|---|---|---|
| numeric_value | Y | string, decimal, or integer | Name of column or Decimal or Integer literal to apply to the function |

For more information on syntax standards, see *Language Documentation Syntax Notes*.

**numeric_value**

Name of the column or numeric literal whose values are used to compute the square root.

> **NOTE:** Negative input values generate null output values.

- Missing input values generate missing results.
- Literal numeric values should not be quoted.
- Multiple columns and wildcards are not supported.

**Usage Notes:**

| Required? | Data Type | Example Value |
|---|---|---|
| Yes | String (column reference) or Integer or Decimal literal | 25 |

## Examples

### Example - Pythagorean Theorem

In this example, you learn how to compute exponentials and square roots on your numeric data.

**Functions**:

| Item | Description |
|---|---|
| POW Function | Computes the value of the first argument raised to the value of the second argument. |
| SQRT Function | Computes the square root of the input parameter. Input value can be a Decimal or Integer literal or a reference to a column containing numeric values. All generated values are non-negative. |

**Source:**

The dataset below contains values for x and y:

| X | Y |
|---|---|
| 3 | 4 |
| 4 | 9 |
| 8 | 10 |
| 30 | 40 |

**Transformation:**

You can use the following transformation to generate values for $z^2$.

**NOTE:** Do not add this step to your recipe right now.

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | (POW(x,2) + POW(y,2)) |
| Parameter: New column name | 'Z' |

You can see how column Z is generated as the sum of squares of the other two columns, which yields $z^2$.

Now, edit the transformation to wrap the value computation in a SQRT function. This step is done to compute the value for *z*, which is the distance between the two points based on the Pythagorean theorem.

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |

| Parameter: Formula | `SQRT((POW(x,2) + POW(y,2)))` |
|---|---|
| Parameter: New column name | `'Z'` |

**Results:**

| X | Y | Z |
|---|---|---|
| 3 | 4 | 5 |
| 4 | 9 | 9.848857801796104 |
| 8 | 10 | 12.806248474865697 |
| 30 | 40 | 50 |

# CEILING Function

Computes the **ceiling** of a value, which is the smallest integer that is greater than the input value. Input can be an Integer, a Decimal, a column reference, or an expression.

**Wrangle vs. SQL:** This function is part of Wrangle , a proprietary data transformation language. Wrangle is not SQL. For more information, see *Wrangle Language*.

## Basic Usage

**Numeric literal example:**

```
ceiling(2.5)
```

**Output:** Generates `3`.

**Expression example:**

```
ceiling(MyValue + 2.5)
```

**Output:** Returns the smallest integer that is greater than the sum of 2.5 and the value in the `MyValue` column.

## Syntax and Arguments

```
ceiling(numeric_value)
```

| Argument | Required? | Data Type | Description |
|---|---|---|---|
| numeric_value | Y | string, decimal, or integer | Name of column or Decimal or Integer literal to apply to the function |

For more information on syntax standards, see *Language Documentation Syntax Notes*.

**numeric_value**

Name of the column, numeric literal, or numeric expression.

- Missing input values generate missing results.
- Literal numeric values should not be quoted. Quoted values are treated as strings.
- Multiple columns and wildcards are not supported.

**Usage Notes:**

| Required? | Data Type | Example Value |
|---|---|---|
| Yes | String (column reference) or Integer or Decimal literal | `2.5` |

## Examples

> **Tip:** For additional examples, see *Common Tasks.*

**Example - Exponential functions**

This example demonstrates how the rounding functions work together.

**Functions**:

| Item | Description |
|---|---|
| FLOOR Function | Computes the largest integer that is not more than the input value. Input can be an Integer, a Decimal, a column reference, or an expression. |
| CEILING Function | Computes the **ceiling** of a value, which is the smallest integer that is greater than the input value. Input can be an Integer, a Decimal, a column reference, or an expression. |
| ROUND Function | Rounds input value to the nearest integer. Input can be an Integer, a Decimal, a column reference, or an expression. Optional second argument can be used to specify the number of digits to which to round. |
| MOD Function | Returns the modulo value, which is the remainder of dividing the first argument by the second argument. Equivalent to the % operator. |

**Source:**

| rowNum | X |
|---|---|
| 1 | -2.5 |
| 2 | -1.2 |
| 3 | 0 |
| 4 | 1 |
| 5 | 1.5 |
| 6 | 2.5 |
| 7 | 3.9 |
| 8 | 4 |
| 9 | 4.1 |
| 10 | 11 |

**Transformation:**

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | FLOOR(X) |
| Parameter: New column name | 'floorX' |

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | CEILING(X) |
| Parameter: New column name | 'ceilingX' |

| Transformation Name | New formula |
|---|---|

| Parameter: Formula type | Single row formula |
|---|---|
| Parameter: Formula | ROUND (X) |
| Parameter: New column name | 'roundX' |

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | (X % 2) |
| Parameter: New column name | 'modX' |

**Results:**

| rowNum | X | modX | roundX | ceilingX | floorX |
|---|---|---|---|---|---|
| 1 | -2.5 | | -2 | -2 | -3 |
| 2 | -1.2 | | -1 | -1 | -2 |
| 3 | 0 | 0 | 0 | 0 | 0 |
| 4 | 1 | 1 | 1 | 1 | 1 |
| 5 | 1.5 | | 2 | 2 | 1 |
| 6 | 2.5 | | 3 | 3 | 2 |
| 7 | 3.9 | | 4 | 4 | 3 |
| 8 | 4 | 0 | 4 | 4 | 4 |
| 9 | 4.1 | | 4 | 5 | 4 |
| 10 | 11 | 1 | 11 | 11 | 11 |

# FLOOR Function

Computes the largest integer that is not more than the input value. Input can be an Integer, a Decimal, a column reference, or an expression.

**Wrangle vs. SQL:** This function is part of Wrangle , a proprietary data transformation language. Wrangle is not SQL. For more information, see *Wrangle Language*.

## Basic Usage

**Numeric literal example:**

```
floor(2.5)
```

**Output:** Returns the value 2.

**Expression example:**

```
floor(MyValue + 2.5)
```

**Output:** Returns the largest integer that is less than the sum of 2.5 and the value in the `MyValue` column.

## Syntax and Arguments

```
floor(numeric_value)
```

| Argument | Required? | Data Type | Description |
|---|---|---|---|
| numeric_value | Y | string, decimal, or integer | Name of column or Decimal or Integer literal to apply to the function |

For more information on syntax standards, see *Language Documentation Syntax Notes*.

**numeric_value**

Name of the column, numeric literal, or numeric expression.

- Missing input values generate missing results.
- Literal numeric values should not be quoted. Quoted values are treated as strings.
- Multiple columns and wildcards are not supported.

**Usage Notes:**

| Required? | Data Type | Example Value |
|---|---|---|
| Yes | String (column reference) or Integer or Decimal literal | `2.5` |

## Examples

> **Tip:** For additional examples, see *Common Tasks*.

## Example - Exponential functions

This example demonstrates how the rounding functions work together.

**Functions**:

| Item | Description |
|---|---|
| FLOOR Function | Computes the largest integer that is not more than the input value. Input can be an Integer, a Decimal, a column reference, or an expression. |
| CEILING Function | Computes the **ceiling** of a value, which is the smallest integer that is greater than the input value. Input can be an Integer, a Decimal, a column reference, or an expression. |
| ROUND Function | Rounds input value to the nearest integer. Input can be an Integer, a Decimal, a column reference, or an expression. Optional second argument can be used to specify the number of digits to which to round. |
| MOD Function | Returns the modulo value, which is the remainder of dividing the first argument by the second argument. Equivalent to the % operator. |

**Source:**

| rowNum | X |
|---|---|
| 1 | -2.5 |
| 2 | -1.2 |
| 3 | 0 |
| 4 | 1 |
| 5 | 1.5 |
| 6 | 2.5 |
| 7 | 3.9 |
| 8 | 4 |
| 9 | 4.1 |
| 10 | 11 |

**Transformation:**

| Transformation Name | New formula |
|---|---|
| **Parameter: Formula type** | Single row formula |
| **Parameter: Formula** | FLOOR(X) |
| **Parameter: New column name** | 'floorX' |

| Transformation Name | New formula |
|---|---|
| **Parameter: Formula type** | Single row formula |
| **Parameter: Formula** | CEILING(X) |
| **Parameter: New column name** | 'ceilingX' |

| Transformation Name | New formula |
|---|---|

| Parameter: Formula type | Single row formula |
|---|---|
| Parameter: Formula | ROUND (X) |
| Parameter: New column name | 'roundX' |

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | (X % 2) |
| Parameter: New column name | 'modX' |

**Results:**

| rowNum | X | modX | roundX | ceilingX | floorX |
|---|---|---|---|---|---|
| 1 | -2.5 | | -2 | -2 | -3 |
| 2 | -1.2 | | -1 | -1 | -2 |
| 3 | 0 | 0 | 0 | 0 | 0 |
| 4 | 1 | 1 | 1 | 1 | 1 |
| 5 | 1.5 | | 2 | 2 | 1 |
| 6 | 2.5 | | 3 | 3 | 2 |
| 7 | 3.9 | | 4 | 4 | 3 |
| 8 | 4 | 0 | 4 | 4 | 4 |
| 9 | 4.1 | | 4 | 5 | 4 |
| 10 | 11 | 1 | 11 | 11 | 11 |

# ROUND Function

**Contents:**

Rounds input value to the nearest integer. Input can be an Integer, a Decimal, a column reference, or an expression. Optional second argument can be used to specify the number of digits to which to round.

- When rounding to nearest integer, decimal values that are $x.5$ or more are rounded to $x+1$.

> **NOTE:** This function changes the actual data of the value. If you just want to change how the data is formatted for display, please use the NUMFORMAT function. See *NUMFORMAT Function*.

**Wrangle vs. SQL:** This function is part of Wrangle , a proprietary data transformation language. Wrangle is not SQL. For more information, see *Wrangle Language*.

## Basic Usage

**Numeric literal example:**

```
round(2.5)
```

**Output:** Rounds the input value to the nearest integer: 3.

**Expression example:**

```
round(MyValue + 2.5)
```

**Output:** Rounds to the nearest integer the sum of 2.5 and the value in the `MyValue` column.

**Numeric literal example:**

```
round(pi(),4)
```

**Output:** Rounds pi to four decimal points: 3.1416.

## Syntax and Arguments

```
round(numeric_value, integer_value)
```

| Argument | Required? | Data Type | Description |
|---|---|---|---|
| numeric_value | Y | string, decimal, or integer | Name of column or Decimal or Integer literal to apply to the function |
| integer_value | N | integer | Number of digits to which to round.<br><br>• Default is $0$, which rounds to the nearest integer.<br>• Negative integer values can be applied. |

For more information on syntax standards, see *Language Documentation Syntax Notes*.

### numeric_value

Name of the column, numeric literal, or numeric expression.

- Missing input values generate missing results.
- Literal numeric values should not be quoted. Quoted values are treated as strings.
- Multiple columns and wildcards are not supported.

**Usage Notes:**

| Required? | Data Type | Example Value |
|---|---|---|
| Yes | String (column reference) or Integer or Decimal literal | `2.5` |

### integer_value

Number of digits to which to round the first argument of the function.

- Positive values values truncate to the right of the decimal point.
- Negative values truncate to the left of the decimal point.
- Missing input values generate missing results.

**Usage Notes:**

| Required? | Data Type | Example Value |
|---|---|---|
| No | Integer literal | `3` |

## Examples

> **Tip:** For additional examples, see *Common Tasks*.

### Example - Exponential functions

This example demonstrates how the rounding functions work together.

**Functions**:

| Item | Description |
|---|---|
| FLOOR Function | Computes the largest integer that is not more than the input value. Input can be an Integer, a Decimal, a column reference, or an expression. |

| | |
|---|---|
| CEILING Function | Computes the **ceiling** of a value, which is the smallest integer that is greater than the input value. Input can be an Integer, a Decimal, a column reference, or an expression. |
| ROUND Function | Rounds input value to the nearest integer. Input can be an Integer, a Decimal, a column reference, or an expression. Optional second argument can be used to specify the number of digits to which to round. |
| MOD Function | Returns the modulo value, which is the remainder of dividing the first argument by the second argument. Equivalent to the % operator. |

### Source:

| rowNum | X |
|---|---|
| 1 | -2.5 |
| 2 | -1.2 |
| 3 | 0 |
| 4 | 1 |
| 5 | 1.5 |
| 6 | 2.5 |
| 7 | 3.9 |
| 8 | 4 |
| 9 | 4.1 |
| 10 | 11 |

### Transformation:

| | |
|---|---|
| **Transformation Name** | New formula |
| **Parameter: Formula type** | Single row formula |
| **Parameter: Formula** | FLOOR(X) |
| **Parameter: New column name** | 'floorX' |

| | |
|---|---|
| **Transformation Name** | New formula |
| **Parameter: Formula type** | Single row formula |
| **Parameter: Formula** | CEILING(X) |
| **Parameter: New column name** | 'ceilingX' |

| | |
|---|---|
| **Transformation Name** | New formula |
| **Parameter: Formula type** | Single row formula |
| **Parameter: Formula** | ROUND (X) |
| **Parameter: New column name** | 'roundX' |

| | |
|---|---|
| **Transformation Name** | New formula |

| Parameter: Formula type | Single row formula |
|---|---|
| Parameter: Formula | (X % 2) |
| Parameter: New column name | 'modX' |

**Results:**

| rowNum | X | modX | roundX | ceilingX | floorX |
|---|---|---|---|---|---|
| 1 | -2.5 | | -2 | -2 | -3 |
| 2 | -1.2 | | -1 | -1 | -2 |
| 3 | 0 | 0 | 0 | 0 | 0 |
| 4 | 1 | 1 | 1 | 1 | 1 |
| 5 | 1.5 | | 2 | 2 | 1 |
| 6 | 2.5 | | 3 | 3 | 2 |
| 7 | 3.9 | | 4 | 4 | 3 |
| 8 | 4 | 0 | 4 | 4 | 4 |
| 9 | 4.1 | | 4 | 5 | 4 |
| 10 | 11 | 1 | 11 | 11 | 11 |

### Example - RANDBETWEEN and PI and ROUND functions

This example illustrates how you can apply functions to generate random numeric data in your dataset.

**Functions**:

| Item | Description |
|---|---|
| RANDBETWEEN Function | Generates a random integer between a low and a high number. Two inputs may be Integer or Decimal types, functions returning these types, or column references. |
| PI Function | The PI function generates the value of pi to 15 decimal places: 3.1415926535897932. |
| ROUND Function | Rounds input value to the nearest integer. Input can be an Integer, a Decimal, a column reference, or an expression. Optional second argument can be used to specify the number of digits to which to round. |
| TRUNC Function | Removes all digits to the right of the decimal point for any value. Optionally, you can specify the number of digits to which to round. Input can be an Integer, a Decimal, a column reference, or an expression. |
| POW Function | Computes the value of the first argument raised to the value of the second argument. |

**Source:**

In the following example, a company produces 10 circular parts, the size of which is measured in each product's radius in inches.

| prodId | radius_in |
|---|---|
| p001 | 1 |
| p002 | 2 |
| p003 | 3 |
| p004 | 4 |

| | |
|---|---|
| p005 | 5 |
| p006 | 6 |
| p007 | 7 |
| p008 | 8 |
| p009 | 9 |
| p010 | 10 |

Based on the above data, the company wants to generate some additional sizing information for these circular parts, including the generation of two points along each part's circumference where quality stress tests can be applied.

**Transformation:**

To begin, you can use the following steps to generate the area and circumference for each product, rounded to three decimal points:

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | ROUND(PI() * (POW(radius_in, 2)), 3) |
| Parameter: New column name | 'area_sqin' |

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | ROUND(PI() * (2 * radius_in), 3) |
| Parameter: New column name | 'circumference_in' |

For quality purposes, the company needs two tests points along the circumference, which are generated by calculating two separate random locations along the circumference. Since the RANDBETWEEN function only calculates using Integer values, you must first truncate the values from circumference_in:

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | TRUNC(circumference_in) |
| Parameter: New column name | 'trunc_circumference_in' |

Then, you can calculate the random points using the following:

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | RANDBETWEEN(0, trunc_circumference_in) |

| Parameter: New column name | 'testPt01_in' |
|---|---|

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | RANDBETWEEN(0, trunc_circumference_in) |
| Parameter: New column name | 'testPt02_in' |

**Results:**

After the `trunc_circumference_in` column is dropped, the data should look similar to the following:

| prodId | radius_in | area_sq_in | circumference_in | testPt01_in | testPt02_in |
|---|---|---|---|---|---|
| p001 | 1 | 3.142 | 6.283 | 5 | 5 |
| p002 | 2 | 12.566 | 12.566 | 3 | 3 |
| p003 | 3 | 28.274 | 18.850 | 13 | 13 |
| p004 | 4 | 50.265 | 25.133 | 24 | 24 |
| p005 | 5 | 78.540 | 31.416 | 0 | 0 |
| p006 | 6 | 113.097 | 37.699 | 15 | 15 |
| p007 | 7 | 153.938 | 43.982 | 11 | 11 |
| p008 | 8 | 201.062 | 50.265 | 1 | 1 |
| p009 | 9 | 254.469 | 56.549 | 29 | 29 |
| p010 | 10 | 314.159 | 62.832 | 21 | 21 |

# TRUNC Function

**Contents:**

---

Removes all digits to the right of the decimal point for any value. Optionally, you can specify the number of digits to which to round. Input can be an Integer, a Decimal, a column reference, or an expression.

**Wrangle vs. SQL:** This function is part of Wrangle , a proprietary data transformation language. Wrangle is not SQL. For more information, see *Wrangle Language*.

## Basic Usage

**Numeric literal example:**

```
trunc(pi())
```

**Output:** Returns the value 3.

**Expression example:**

```
trunc(length_in * length_in, 2)
```

**Output:** Returns the square of the values in `length_in`, truncated to two decimal points.

## Syntax and Arguments

```
trunc(numeric_value,integer_value)
```

| Argument | Required? | Data Type | Description |
|---|---|---|---|
| numeric_value | Y | string, decimal, or integer | Name of column or Decimal or Integer literal to apply to the function |
| integer_value | N | integer | Number of digits to which to truncate.<br><br>• Default is 0, which truncates to the nearest integer.<br>• Negative integer values can be applied. |

For more information on syntax standards, see *Language Documentation Syntax Notes*.

**numeric_value**

Name of the column, numeric literal, or numeric expression.

- Missing input values generate missing results.

- Literal numeric values should not be quoted. Quoted values are treated as strings.
- Multiple columns and wildcards are not supported.

**Usage Notes:**

| Required? | Data Type | Example Value |
|---|---|---|
| Yes | String (column reference) or Integer or Decimal literal | `2.5` |

**integer_value**

Number of digits to which to round the first argument of the function.

- Positive values values truncate to the right of the decimal point.
- Negative values truncate to the left of the decimal point.
- Missing input values generate missing results.

**Usage Notes:**

| Required? | Data Type | Example Value |
|---|---|---|
| No | Integer literal | 3 |

## Examples

> **Tip:** For additional examples, see *Common Tasks*.

**Example - Basic TRUNC**

**Source:**

| RowId | myVal |
|---|---|
| r01 | 1.2345 |
| r02 | -1.2345 |
| r03 | 100.000 |
| r04 | 10.1 |
| r05 | 50.029 |

**Transformation:**

| Transformation Name | `New formula` |
|---|---|
| **Parameter: Formula type** | `Single row formula` |
| **Parameter: Formula** | `trunc(myVal)` |
| **Parameter: New column name** | `'trunc_myVal'` |

| Transformation Name | `New formula` |
|---|---|

| Parameter: Formula type | `Single row formula` |
|---|---|
| Parameter: Formula | `trunc(myVal,2)` |
| Parameter: New column name | `'trunc_myVal2'` |

| Transformation Name | `New formula` |
|---|---|
| Parameter: Formula type | `Single row formula` |
| Parameter: Formula | `trunc(myVal,2)` |
| Parameter: New column name | `'trunc_myVal_2'` |

**Results:**

| RowId | myVal | trunc_myVal | trunc_myVal2 | trunc_myVal_2 |
|---|---|---|---|---|
| r01 | 1.2345 | 1 | 1.23 | 0 |
| r02 | -1.2345 | -1 | -1.23 | 0 |
| r03 | 100.000 | 100 | 100.00 | 100 |
| r04 | 10.1 | 10 | 10.1 | 0 |
| r05 | 50.029 | 50 | 50.02 | 0 |

### Example - RANDBETWEEN, PI, and TRUNC functions

This example illustrates how you can apply functions to generate random numeric data in your dataset.

**Functions**:

| Item | Description |
|---|---|
| RANDBETWEEN Function | Generates a random integer between a low and a high number. Two inputs may be Integer or Decimal types, functions returning these types, or column references. |
| PI Function | The `PI` function generates the value of pi to 15 decimal places: 3.1415926535897932. |
| ROUND Function | Rounds input value to the nearest integer. Input can be an Integer, a Decimal, a column reference, or an expression. Optional second argument can be used to specify the number of digits to which to round. |
| TRUNC Function | Removes all digits to the right of the decimal point for any value. Optionally, you can specify the number of digits to which to round. Input can be an Integer, a Decimal, a column reference, or an expression. |
| POW Function | Computes the value of the first argument raised to the value of the second argument. |

**Source:**

In the following example, a company produces 10 circular parts, the size of which is measured in each product's radius in inches.

| prodId | radius_in |
|---|---|
| p001 | 1 |
| p002 | 2 |
| p003 | 3 |

| p004 | 4 |
|---|---|
| p005 | 5 |
| p006 | 6 |
| p007 | 7 |
| p008 | 8 |
| p009 | 9 |
| p010 | 10 |

Based on the above data, the company wants to generate some additional sizing information for these circular parts, including the generation of two points along each part's circumference where quality stress tests can be applied.

**Transformation:**

To begin, you can use the following steps to generate the area and circumference for each product, rounded to three decimal points:

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | ROUND(PI() * (POW(radius_in, 2)), 3) |
| Parameter: New column name | 'area_sqin' |

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | ROUND(PI() * (2 * radius_in), 3) |
| Parameter: New column name | 'circumference_in' |

For quality purposes, the company needs two tests points along the circumference, which are generated by calculating two separate random locations along the circumference. Since the RANDBETWEEN function only calculates using Integer values, you must first truncate the values from circumference_in:

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | TRUNC(circumference_in) |
| Parameter: New column name | 'trunc_circumference_in' |

Then, you can calculate the random points using the following:

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | RANDBETWEEN(0, trunc_circumference_in) |

| Parameter: New column name | 'testPt01_in' |
|---|---|

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | RANDBETWEEN(0, trunc_circumference_in) |
| Parameter: New column name | 'testPt02_in' |

**Results:**

After the `trunc_circumference_in` column is dropped, the data should look similar to the following:

| prodId | radius_in | area_sq_in | circumference_in | testPt01_in | testPt02_in |
|---|---|---|---|---|---|
| p001 | 1 | 3.142 | 6.283 | 5 | 5 |
| p002 | 2 | 12.566 | 12.566 | 3 | 3 |
| p003 | 3 | 28.274 | 18.850 | 13 | 13 |
| p004 | 4 | 50.265 | 25.133 | 24 | 24 |
| p005 | 5 | 78.540 | 31.416 | 0 | 0 |
| p006 | 6 | 113.097 | 37.699 | 15 | 15 |
| p007 | 7 | 153.938 | 43.982 | 11 | 11 |
| p008 | 8 | 201.062 | 50.265 | 1 | 1 |
| p009 | 9 | 254.469 | 56.549 | 29 | 29 |
| p010 | 10 | 314.159 | 62.832 | 21 | 21 |

# NUMVALUE Function

**Contents:**

---

Converts a string formatted as a number into an Integer or Decimal value by parsing out the specified decimal and group separators. A string or a function returning formatted numbers of String type or a column containing formatted numbers of string type can be inputs.

You can use this function to convert String values that have locale-specific formatting to locale-independent values of Integer or Decimal type by removing the formatting separators.

- If the source value does not include a valid input for this function, a missing value is returned.
- This function supports negative input values.
- Supports input for multiple locale types and returns the output of the appropriate numeric type.

You can use decimal separators and grouping separators when working with other currency formats. More information is below.

> **NOTE**: If the decimal separator and group separator arguments are not specified, then a null value is returned.

**Wrangle vs. SQL:** This function is part of Wrangle , a proprietary data transformation language. Wrangle is not SQL. For more information, see *Wrangle Language*.

## Basic Usage

```
Numvalue(<code>Discount</code>, &quot;<code>,</code>&quot; ,&quot;<code>.&quot;</code>)
```

**Output:** Returns the values from the `Discount` column by converting the formatted number to a numeric value using decimal and group separators. For example, if the `Discount` column has values in percentage, then it is converted into the corresponding numeric value. Example: 10% to 0.10.

> **NOTE:** If multiple percent signs are used in a column, then this function returns the decimal values accordingly. For example, an input of `9%%` returns `0.0009`.

> **NOTE**: This function removes any currency indicators. For example, `£ 100` as an input value returns `100`.

## Syntax and Arguments

```
<code class="listtype-code listindent1 list-code1 lang-bash">Numvalue(formattedNumber,
[&quot;grouping_separator&quot;], [&quot;decimal_separator&quot;])</code>
```

| Argument | Required? | Data Type | Description |
|---|---|---|---|
| formattedNumber | Y | string | A string, a column of strings, or a function returning a string |
| grouping_separator | Y | string | A grouping representing grouping separator. By default, comma (,) is used as the grouping separator. |
| decimal_separator | Y | string | A string used to separate the integer and fractional part of the result. If not specified, then a null value is returned. |

For more information on syntax standards, see *Language Documentation Syntax Notes.*

### formatted_number

Literal string value, a function that returns a string value, or the name of a column containing string values to be converted into a numeric value.

- Multiple columns and wildcards are not supported.

**Usage Notes:**

| Required? | Data Type | Example Value |
|---|---|---|
| Yes | string, a column of strings, or a function returning a formatted number string or column of strings | Discount |

### grouping_separator

The string used to group a set of digits in the input values. Separators must be enclosed with double quotes (") or single quotes (').  For example, a comma  ( , )  is used as a grouping separator in the U.S.A. ("10,000"), whereas space is used in France ("10 000").

> **NOTE**: When you provide invalid separators or wrong separators, you may get an error in the Formula column.

**Usage Notes:**

| Required? | Data Type | Example Value |
|---|---|---|
| Yes | String literal | ',' |

### decimal_separator

The string used to indicate the decimal point in the input values. Separators must be enclosed with double quotes ( " ) or single quotes ( ' ).

A decimal separator is used to separate  the fractional  part of a number written in decimal form. For example, a period (.) is used as a decimal separator in the U.S.A. ("1234.12"), whereas comma (,) is used in France ("1234,12").

**Usage Notes:**

| Required? | Data Type | Example Value |
|---|---|---|
| Yes | String literal | '.' |

| Grouping Separator | Decimal Separator | Example Locale |
|---|---|---|
| Comma ( , ) | Period ( . ) | U.S locale |
| Period ( . ) | Comma ( , ) | Spanish locale |
| Space | Comma ( , ) | French locale |

## Examples

> **Tip:** For additional examples, see *Common Tasks*.

### Example - formatting price and percentages

This example shows how to convert the formatted number of string type into a numeric value. The following table shows different types of products and their total sales in the UK region. From this example, you must convert the total sales value for the U.S region.

First, you must convert the global currency formats into a generic numeric value, then proceed with the conversion calculation.

**Source:**

| Products | Total_Sales_UK |
|---|---|
| Baby Foods | £ 100.00 |
| Medicines | £ 150.00 |
| Groceries | £ 200.00 |
| Kitchen Supplies | £ 25.00 |
| Cosmetics | £ 250.00 |
| Snacks | £ 50.00 |

**Transformation**:

The first transformation is to convert the `Total_Sales_UK` column into a numeric value. In this case, when a `Numvalue` function is applied, the value with the currency symbols returns only the numeric value. You can see the currency symbol is removed when using the `Numvalue` function.

| Transformation Name | Edit column with formula |
|---|---|
| Parameter: columns | Total_Sales_UK |
| Parameter: Formula | NUMVALUE (Total_Sales_UK, ",",".") |

The second step consists of converting the U.K Pounds (£) to U.S Dollars ($) for better computations. In this step, you multiply the `Total_Sales_UK` column with the ratio conversion rate. Let's say the current conversion rate is 1.36, then multiply the U.K Pounds with 1.36 to convert to U.S Dollars. As a part of this transformation, the `Total_Sales_US` column is created.

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | Total_Sales_UK * 1.36 |
| Parameter: New column name | Total_Sales_US |

After you get the U.S conversion, you can format the values using the `Numformat` function. The `Numformat` function formats a numeric set of values according to the specified number formatting.

| Transformation Name | Edit column with formula |
|---|---|
| Parameter: columns | Total_Sales_US |
| Parameter: Formula | NUMFORMAT(Total_Sales_US, '$###,###.00', ',', '.') |

**Results**

The output data should look like the following:

| Products | Total_Sales_UK | Total_Sales_US |
|---|---|---|
| Baby Foods | £  100.00 | $136.00 |
| Medicines | £  150.00 | $204.00 |
| Groceries | £  200 .00 | $272.00 |
| Kitchen Supplies | £  25 .00 | $34. 00 |
| Cosmetics | £  250 .00 | $340. 00 |
| Snacks | £  50 .00 | $68. 00 |

# Trigonometry Functions

The following trigonometric functions can be applied to your transformations.

# SIN Function

Computes the sine of an input value for an angle measured in radians. The value can be a Decimal or Integer literal or a reference to a column containing numeric values.
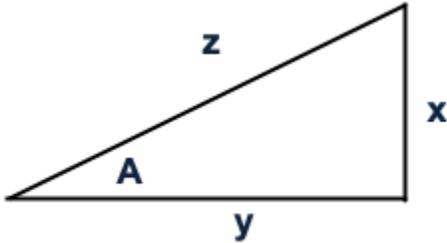


In the above, the sine of angle A is computed as the following:

```
sin(A) = x/z
```

The cosecant of angle A is 1/sin(A), or:

```
csc(A) = 1/sin(A) = z/x
```

You can convert from degrees to radians. For more information, see *RADIANS Function*.

**Wrangle vs. SQL:** This function is part of Wrangle , a proprietary data transformation language. Wrangle is not SQL. For more information, see *Wrangle Language*.

## Basic Usage

**Numeric literal example:**

```
round(sin(radians(30)),3)
```

**Output:** Returns the computation of the sine of a 30-degree angle, which is converted to radians before being passed to the SIN function. The output value is rounded to three decimals: `0.500`.

**Column reference example:**

```
sin(X)
```

**Output:** Returns  the sine of the radians values in `X` column.

## Syntax and Arguments

```
sin(numeric_value)
```

| Argument | Required? | Data Type | Description |
|---|---|---|---|

| | | | |
|---|---|---|---|
| numeric_val ue | Y | string, decimal, or integer | Name of column, Decimal or Integer literal, or function returning those types to apply to the function |

For more information on syntax standards, see *Language Documentation Syntax Notes*.

### numeric_value

Name of the column, Integer or Decimal literal, or function returning that data type to apply to the function.

- Missing input values generate missing results.
- Literal numeric values should not be quoted. Quoted values are treated as strings.
- Multiple columns and wildcards are not supported.

### Usage Notes:

| Required? | Data Type | Example Value |
|---|---|---|
| Yes | String (column reference) or Integer or Decimal literal | `0.5` |

## Examples

> **Tip:** For additional examples, see *Common Tasks*.

### Example - Trigonometry functions

This example illustrates how to apply basic trigonometric functions to your transformations.

### Functions:

| Item | Description |
|---|---|
| SIN Function | Computes the sine of an input value for an angle measured in radians. The value can be a Decimal or Integer literal or a reference to a column containing numeric values. |
| COS Function | Computes the cosine of an input value for an angle measured in radians. The value can be a Decimal or Integer literal or a reference to a column containing numeric values. |
| TAN Function | Computes the tangent of an input value for an angle measured in radians. The value can be a Decimal or Integer literal or a reference to a column containing numeric values. |

The following functions are computed using the above functions:

- **Cotangent.** Computed as 1/TAN.
- **Secant.** Computed as 1/COS.
- **Cosecant.** Computed as 1/SIN.

Also:

| Item | Description |
|---|---|
| ROUND Function | Rounds input value to the nearest integer. Input can be an Integer, a Decimal, a column reference, or an expression. Optional second argument can be used to specify the number of digits to which to round. |
| RADIANS Function | Computes the radians of an input value measuring degrees of an angle. The value can be a Decimal or Integer literal or a reference to a column containing numeric values. |

### Source:

In the following sample, input values are in degrees:

| X |
|---|
| -30 |
| 0 |
| 30 |
| 45 |
| 60 |
| 90 |
| 120 |
| 135 |
| 180 |

**Transformation:**

In this example, all values are rounded to three decimals for clarity.

First, the above values in degrees must be converted to radians.

| Transformation Name | New formula |
|---|---|
| **Parameter: Formula type** | Single row formula |
| **Parameter: Formula** | round(radians(X), 3) |
| **Parameter: New column name** | 'rX' |

Sine:

| Transformation Name | New formula |
|---|---|
| **Parameter: Formula type** | Single row formula |
| **Parameter: Formula** | round(sin(rX), 3) |
| **Parameter: New column name** | 'SINrX' |

Cosine:

| Transformation Name | New formula |
|---|---|
| **Parameter: Formula type** | Single row formula |
| **Parameter: Formula** | round(cos(rX), 3) |
| **Parameter: New column name** | 'COSrX' |

Tangent:

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | round(tan(rX), 3) |
| Parameter: New column name | 'TANrX' |

Cotangent:

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | round(1 / tan(rX), 3) |
| Parameter: New column name | 'COTrX' |

Secant:

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | round(1 / cos(rX), 3) |
| Parameter: New column name | 'SECrX' |

Cosecant:

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | round(1 / sin(rX), 3) |
| Parameter: New column name | 'CSCrX' |

**Results:**

| X | rX | COTrX | SECrX | CSCrX | TANrX | COSrX | SINrX |
|---|---|---|---|---|---|---|---|
| -30 | -0.524 | -1.73 | 1.155 | -1.999 | -0.578 | 0.866 | -0.5 |
| 0 | 0 | *null* | 1 | *null* | 0 | 1 | 0 |
| 30 | 0.524 | 1.73 | 1.155 | 1.999 | 0.578 | 0.866 | 0.5 |
| 45 | 0.785 | 1.001 | 1.414 | 1.415 | 0.999 | 0.707 | 0.707 |
| 60 | 1.047 | 0.578 | 1.999 | 1.155 | 1.731 | 0.5 | 0.866 |
| 90 | 1.571 | 0 | -4909.826 | 1 | -4909.826 | 0 | 1 |
| 120 | 2.094 | -0.577 | -2.001 | 1.154 | -1.734 | -0.5 | 0.866 |
| 135 | 2.356 | -1 | -1.414 | 1.414 | -1 | -0.707 | 0.707 |
| 180 | 3.142 | 2454.913 | -1 | -2454.913 | 0 | -1 | 0 |

# COS Function

Computes the cosine of an input value for an angle measured in radians. The value can be a Decimal or Integer literal or a reference to a column containing numeric values.
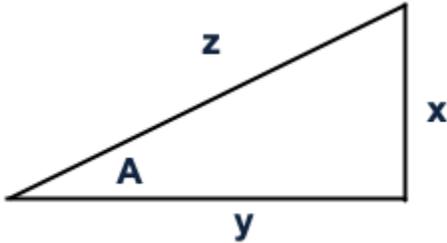


In the above, the cosine of angle A is computed as the following:

```
cos(A) = y/z
```

The secant of angle A is 1/cos(A), or:

```
sec(A) = 1/cos(A) = z/y
```

You can convert from degrees to radians. For more information, see *RADIANS Function*.

**Wrangle vs. SQL:** This function is part of Wrangle , a proprietary data transformation language. Wrangle is not SQL. For more information, see *Wrangle Language*.

## Basic Usage

**Numeric literal example:**

```
round(cos(radians(30)),3)
```

**Output:** Returns the computation of the cosine of a 30-degree angle, which is converted to radians before being passed to the COS function. The output value is rounded to three decimals: `0.866`.

**Column reference example:**

```
cos(X)
```

**Output:** Returns the cosine of the radians values in `X` column.

## Syntax and Arguments

```
cos(numeric_value)
```

| Argument | Required? | Data Type | Description |
|---|---|---|---|

| numeric_value | Y | string, decimal, or integer | Name of column, Decimal or Integer literal, or function returning those types to apply to the function |
|---|---|---|---|

For more information on syntax standards, see *Language Documentation Syntax Notes*.

### numeric_value

Name of the column, Integer or Decimal literal, or function returning that data type to apply to the function.

- Missing input values generate missing results.
- Literal numeric values should not be quoted. Quoted values are treated as strings.
- Multiple columns and wildcards are not supported.

### Usage Notes:

| Required? | Data Type | Example Value |
|---|---|---|
| Yes | String (column reference) or Integer or Decimal literal | `0.5` |

## Examples

> **Tip:** For additional examples, see *Common Tasks*.

### Example - Trigonometry functions

This example illustrates how to apply basic trigonometric functions to your transformations.

**Functions**:

| Item | Description |
|---|---|
| SIN Function | Computes the sine of an input value for an angle measured in radians. The value can be a Decimal or Integer literal or a reference to a column containing numeric values. |
| COS Function | Computes the cosine of an input value for an angle measured in radians. The value can be a Decimal or Integer literal or a reference to a column containing numeric values. |
| TAN Function | Computes the tangent of an input value for an angle measured in radians. The value can be a Decimal or Integer literal or a reference to a column containing numeric values. |

The following functions are computed using the above functions:

- **Cotangent.** Computed as 1/TAN.
- **Secant.** Computed as 1/COS.
- **Cosecant.** Computed as 1/SIN.

Also:

| Item | Description |
|---|---|
| ROUND Function | Rounds input value to the nearest integer. Input can be an Integer, a Decimal, a column reference, or an expression. Optional second argument can be used to specify the number of digits to which to round. |
| RADIANS Function | Computes the radians of an input value measuring degrees of an angle. The value can be a Decimal or Integer literal or a reference to a column containing numeric values. |

**Source:**

In the following sample, input values are in degrees:

| X |
|---|
| -30 |
| 0 |
| 30 |
| 45 |
| 60 |
| 90 |
| 120 |
| 135 |
| 180 |

**Transformation:**

In this example, all values are rounded to three decimals for clarity.

First, the above values in degrees must be converted to radians.

| **Transformation Name** | New formula |
|---|---|
| **Parameter: Formula type** | Single row formula |
| **Parameter: Formula** | round(radians(X), 3) |
| **Parameter: New column name** | 'rX' |

Sine:

| **Transformation Name** | New formula |
|---|---|
| **Parameter: Formula type** | Single row formula |
| **Parameter: Formula** | round(sin(rX), 3) |
| **Parameter: New column name** | 'SINrX' |

Cosine:

| **Transformation Name** | New formula |
|---|---|
| **Parameter: Formula type** | Single row formula |
| **Parameter: Formula** | round(cos(rX), 3) |
| **Parameter: New column name** | 'COSrX' |

Tangent:

| Transformation Name | New formula |
| --- | --- |
| Parameter: Formula type | Single row formula |
| Parameter: Formula | round(tan(rX), 3) |
| Parameter: New column name | 'TANrX' |

Cotangent:

| Transformation Name | New formula |
| --- | --- |
| Parameter: Formula type | Single row formula |
| Parameter: Formula | round(1 / tan(rX), 3) |
| Parameter: New column name | 'COTrX' |

Secant:

| Transformation Name | New formula |
| --- | --- |
| Parameter: Formula type | Single row formula |
| Parameter: Formula | round(1 / cos(rX), 3) |
| Parameter: New column name | 'SECrX' |

Cosecant:

| Transformation Name | New formula |
| --- | --- |
| Parameter: Formula type | Single row formula |
| Parameter: Formula | round(1 / sin(rX), 3) |
| Parameter: New column name | 'CSCrX' |

**Results:**

| X | rX | COTrX | SECrX | CSCrX | TANrX | COSrX | SINrX |
| --- | --- | --- | --- | --- | --- | --- | --- |
| -30 | -0.524 | -1.73 | 1.155 | -1.999 | -0.578 | 0.866 | -0.5 |
| 0 | 0 | *null* | 1 | *null* | 0 | 1 | 0 |
| 30 | 0.524 | 1.73 | 1.155 | 1.999 | 0.578 | 0.866 | 0.5 |
| 45 | 0.785 | 1.001 | 1.414 | 1.415 | 0.999 | 0.707 | 0.707 |
| 60 | 1.047 | 0.578 | 1.999 | 1.155 | 1.731 | 0.5 | 0.866 |
| 90 | 1.571 | 0 | -4909.826 | 1 | -4909.826 | 0 | 1 |
| 120 | 2.094 | -0.577 | -2.001 | 1.154 | -1.734 | -0.5 | 0.866 |
| 135 | 2.356 | -1 | -1.414 | 1.414 | -1 | -0.707 | 0.707 |
| 180 | 3.142 | 2454.913 | -1 | -2454.913 | 0 | -1 | 0 |

# TAN Function

Computes the tangent of an input value for an angle measured in radians. The value can be a Decimal or Integer literal or a reference to a column containing numeric values.
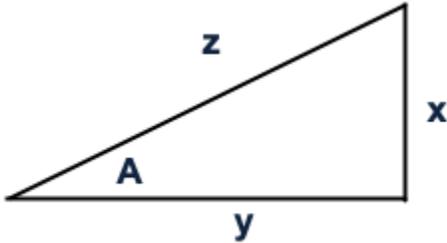


In the above, the tangent of angle A is computed as the following:

```
tan(A) = x/y
```

The cotangent of angle A is 1/tan(A), or:

```
cot(A) = 1/tan(A) = y/x
```

You can convert from degrees to radians. For more information, see *RADIANS Function*.

**Wrangle vs. SQL:** This function is part of Wrangle , a proprietary data transformation language. Wrangle is not SQL. For more information, see *Wrangle Language*.

## Basic Usage

**Numeric literal example:**

```
round(tan(radians(30)),3)
```

**Output:** Returns the computation of the tangent of a 30-degree angle, which is converted to radians before being passed to the `tan` function. The output value is rounded to three decimals: `0.577`.

**Column reference example:**

```
tan(X)
```

**Output:** Returns the tangent of the radians values in `X` column.

## Syntax and Arguments

```
tan(numeric_value)
```

| Argument | Required? | Data Type | Description |
|----------|-----------|-----------|-------------|

| | | | |
|---|---|---|---|
| numeric_val ue | Y | string, decimal, or integer | Name of column, Decimal or Integer literal, or function returning those types to apply to the function |

For more information on syntax standards, see *Language Documentation Syntax Notes*.

### numeric_value

Name of the column, Integer or Decimal literal, or function returning that data type to apply to the function.

- Missing input values generate missing results.
- Literal numeric values should not be quoted. Quoted values are treated as strings.
- Multiple columns and wildcards are not supported.

### Usage Notes:

| Required? | Data Type | Example Value |
|---|---|---|
| Yes | String (column reference) or Integer or Decimal literal | `0.5` |

## Examples

> **Tip:** For additional examples, see *Common Tasks*.

### Example - Trigonometry functions

This example illustrates how to apply basic trigonometric functions to your transformations.

**Functions**:

| Item | Description |
|---|---|
| SIN Function | Computes the sine of an input value for an angle measured in radians. The value can be a Decimal or Integer literal or a reference to a column containing numeric values. |
| COS Function | Computes the cosine of an input value for an angle measured in radians. The value can be a Decimal or Integer literal or a reference to a column containing numeric values. |
| TAN Function | Computes the tangent of an input value for an angle measured in radians. The value can be a Decimal or Integer literal or a reference to a column containing numeric values. |

The following functions are computed using the above functions:

- **Cotangent.** Computed as 1/TAN.
- **Secant.** Computed as 1/COS.
- **Cosecant.** Computed as 1/SIN.

Also:

| Item | Description |
|---|---|
| ROUND Function | Rounds input value to the nearest integer. Input can be an Integer, a Decimal, a column reference, or an expression. Optional second argument can be used to specify the number of digits to which to round. |
| RADIANS Function | Computes the radians of an input value measuring degrees of an angle. The value can be a Decimal or Integer literal or a reference to a column containing numeric values. |

**Source:**

In the following sample, input values are in degrees:

| X |
|---|
| -30 |
| 0 |
| 30 |
| 45 |
| 60 |
| 90 |
| 120 |
| 135 |
| 180 |

**Transformation:**

In this example, all values are rounded to three decimals for clarity.

First, the above values in degrees must be converted to radians.

| Transformation Name | New formula |
|---|---|
| **Parameter: Formula type** | Single row formula |
| **Parameter: Formula** | round(radians(X), 3) |
| **Parameter: New column name** | 'rX' |

Sine:

| Transformation Name | New formula |
|---|---|
| **Parameter: Formula type** | Single row formula |
| **Parameter: Formula** | round(sin(rX), 3) |
| **Parameter: New column name** | 'SINrX' |

Cosine:

| Transformation Name | New formula |
|---|---|
| **Parameter: Formula type** | Single row formula |
| **Parameter: Formula** | round(cos(rX), 3) |
| **Parameter: New column name** | 'COSrX' |

Tangent:

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | round(tan(rX), 3) |
| Parameter: New column name | 'TANrX' |

Cotangent:

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | round(1 / tan(rX), 3) |
| Parameter: New column name | 'COTrX' |

Secant:

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | round(1 / cos(rX), 3) |
| Parameter: New column name | 'SECrX' |

Cosecant:

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | round(1 / sin(rX), 3) |
| Parameter: New column name | 'CSCrX' |

**Results:**

| X | rX | COTrX | SECrX | CSCrX | TANrX | COSrX | SINrX |
|---|---|---|---|---|---|---|---|
| -30 | -0.524 | -1.73 | 1.155 | -1.999 | -0.578 | 0.866 | -0.5 |
| 0 | 0 | *null* | 1 | *null* | 0 | 1 | 0 |
| 30 | 0.524 | 1.73 | 1.155 | 1.999 | 0.578 | 0.866 | 0.5 |
| 45 | 0.785 | 1.001 | 1.414 | 1.415 | 0.999 | 0.707 | 0.707 |
| 60 | 1.047 | 0.578 | 1.999 | 1.155 | 1.731 | 0.5 | 0.866 |
| 90 | 1.571 | 0 | -4909.826 | 1 | -4909.826 | 0 | 1 |
| 120 | 2.094 | -0.577 | -2.001 | 1.154 | -1.734 | -0.5 | 0.866 |
| 135 | 2.356 | -1 | -1.414 | 1.414 | -1 | -0.707 | 0.707 |
| 180 | 3.142 | 2454.913 | -1 | -2454.913 | 0 | -1 | 0 |

# ASIN Function

For input values between -1 and 1 inclusive, this function returns the angle in radians whose sine value is the input. This function is the inverse of the sine function. The value can be a Decimal or Integer literal or a reference to a column containing numeric values.

> **NOTE:** While this function returns values outside of the range -1 <= x <= 1, those values are not considered valid.

For more information on the sine function, see *SIN Function*.

**Arc cosecant:**

The arc secant function is computed as follows:

| Input Range | Output computation |
|---|---|
| x<=-1 | `asin(1/x)` |
| x>=1 | `asin(1/x)` |
| -1<x<1 | Not valid |

**Wrangle vs. SQL:** This function is part of Wrangle , a proprietary data transformation language. Wrangle is not SQL. For more information, see *Wrangle Language*.

## Basic Usage

**Numeric literal example:**

```
asin(0.5)
```

**Output:** Returns the computation of the arc sine of 0.5. Output value is in radians.

**Column reference example:**

```
asin(X)
```

**Output:** Returns the arc sine of the values in `X` column.

## Syntax and Arguments

```
asin(numeric_value)
```

| Argument | Required? | Data Type | Description |
|---|---|---|---|
| numeric_value | Y | string, decimal, or integer | Name of column, Decimal or Integer literal, or function returning those types to apply to the function |

For more information on syntax standards, see *Language Documentation Syntax Notes*.

**numeric_value**

Name of the column, Integer or Decimal literal, or function returning that data type to apply to the function.

- Missing input values generate missing results.
- Literal numeric values should not be quoted. Quoted values are treated as strings.
- Multiple columns and wildcards are not supported.

**Usage Notes:**

| Required? | Data Type | Example Value |
|---|---|---|
| Yes | String (column reference) or Integer or Decimal literal | `10` |

## Examples

> **Tip:** For additional examples, see *Common Tasks*.

### Example - Trigonometry Arc functions

This example illustrates how to apply the inverse trigonometric (Arc) functions to your transformations.

**Functions:**

| Item | Description |
|---|---|
| ASIN Function | For input values between -1 and 1 inclusive, this function returns the angle in radians whose sine value is the input. This function is the inverse of the sine function. The value can be a Decimal or Integer literal or a reference to a column containing numeric values. |
| ACOS Function | For input values between -1 and 1 inclusive, this function returns the angle in radians whose cosine value is the input. This function is the inverse of the cosine function. The value can be a Decimal or Integer literal or a reference to a column containing numeric values. |
| ATAN Function | For input values between -1 and 1 inclusive, this function returns the angle in radians whose tangent value is the input. This function is the inverse of the tangent function. The value can be a Decimal or Integer literal or a reference to a column containing numeric values. |

Also:

| Item | Description |
|---|---|
| ROUND Function | Rounds input value to the nearest integer. Input can be an Integer, a Decimal, a column reference, or an expression. Optional second argument can be used to specify the number of digits to which to round. |
| DEGREES Function | Computes the degrees of an input value measuring the radians of an angle. The value can be a Decimal or Integer literal or a reference to a column containing numeric values. |

> **NOTE:** These functions are valid over specific ranges.

The following functions are computed using the above functions.

- Arccotangent. Computed using ATAN function. See below.
- Arcsecant. Computed using ACOS function. See below.
- Arccosecant. Computed using ASIN function. See below.

**Source:**

In the following sample, input values are in radians. In this example, all values are rounded to two decimals for clarity.

| Y |
|---|
| -1.00 |
| -0.75 |
| -0.50 |
| 0.00 |
| 0.50 |
| 0.75 |
| 1.00 |

**Transformation:**

Arcsine:

Valid over the range (-1 <= Y <= 1)

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | round(degrees(asin(Y)), 2) |
| Parameter: New column name | 'asinY' |

Arccosine:

Valid over the range (-1 <= Y <= 1)

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | round(degrees(acos(Y)), 2) |
| Parameter: New column name | 'acosY' |

Arctangent:

Valid over the range (-1 <= Y <= 1)

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | round(degrees(atan(Y)), 2) |
| Parameter: New column name | 'atanY' |

Arccosecant:

This function is valid over a set of ranged inputs, so you can use a conditional column for the computation.

| Transformation Name | Conditional column |
|---|---|
| Parameter: Condition type | if...then...else |
| Parameter: If | (Y <= -1) \|\| (Y >= 1) |
| Parameter: Then | round(degrees(asin(divide(1, Y))), 2) |
| Parameter: New column name | 'acscY' |

Arcsecant:

Same set of ranged inputs apply to this function.

| Transformation Name | Conditional column |
|---|---|
| Parameter: Condition type | if...then...else |
| Parameter: If | (Y <= -1) \|\| (Y >= 1) |
| Parameter: Then | round(degrees(acos(divide(1, Y))), 2) |
| Parameter: New column name | 'asecY' |

Arccotangent:

For this function, the two different ranges of inputs have different computations, so an `else` condition is added to the transformation.

| Transformation Name | Conditional column |
|---|---|
| Parameter: Condition type | if...then...else |
| Parameter: If | Y > 0 |
| Parameter: Then | round(degrees(atan(divide(1, Y))), 2) |
| Parameter: Else | round(degrees(atan(divide(1, Y)) + pi()), 2) |
| Parameter: New column name | 'acotY' |

**Results:**

| Y | acotY | asecY | acscY | atanY | acosY | asinY |
|---|---|---|---|---|---|---|
| -1.00 | -41.86 | 180.00 | -90.00 | -45.00 | 180.00 | -90.00 |
| -0.75 | -49.99 | *null* | *null* | -37.00 | 139.00 | -49.00 |
| -0.50 | -60.29 | *null* | *null* | -27.00 | 120.00 | -30.00 |
| 0.00 | *null* | *null* | *null* | 0.00 | 90.00 | 0.00 |
| 0.50 | 63.44 | *null* | *null* | 27.00 | 60.00 | 30.00 |

| 0.75 | 53.13 | *null* | *null* | 37.00 | 41.00 | 49.00 |
| 1.00 | 45.00 | 0.00 | 90.00 | 45.00 | 0.00 | 90.00 |

# ACOS Function

For input values between -1 and 1 inclusive, this function returns the angle in radians whose cosine value is the input. This function is the inverse of the cosine function. The value can be a Decimal or Integer literal or a reference to a column containing numeric values.

> **NOTE:** While this function returns values outside of the range -1 <= x <= 1, those values are not considered valid.

For more information on the cosine function, see *COS Function.*

**Arc secant:**

The arc secant function is computed as follows:

| Input Range | Output computation |
|---|---|
| x<=-1 | `acos(1/x)` |
| x>=1 | `acos(1/x)` |
| -1<x<1 | Not valid |

**Wrangle vs. SQL:** This function is part of Wrangle , a proprietary data transformation language. Wrangle is not SQL. For more information, see *Wrangle Language.*

## Basic Usage

**Numeric literal example:**

```
acos(0.5)
```

**Output:** Returns the computation of the arc cosine of 0.5. Output value is in radians.

**Column reference example:**

```
acos(X)
```

**Output:** Returns the arc cosine of the values in `X` column.

## Syntax and Arguments

```
acos(numeric_value)
```

| Argument | Required? | Data Type | Description |
|---|---|---|---|
| numeric_val ue | Y | string, decimal, or integer | Name of column, Decimal or Integer literal, or function returning those types to apply to the function |

For more information on syntax standards, see *Language Documentation Syntax Notes.*

**numeric_value**

Name of the column, Integer or Decimal literal, or function returning that data type to apply to the function.

- Missing input values generate missing results.
- Literal numeric values should not be quoted. Quoted values are treated as strings.
- Multiple columns and wildcards are not supported.

**Usage Notes:**

| Required? | Data Type | Example Value |
|---|---|---|
| Yes | String (column reference) or Integer or Decimal literal | `0.5` |

## Examples

> **Tip:** For additional examples, see *Common Tasks*.

### Example - Trigonometry Arc functions

This example illustrates how to apply the inverse trigonometric (Arc) functions to your transformations.

**Functions:**

| Item | Description |
|---|---|
| ASIN Function | For input values between -1 and 1 inclusive, this function returns the angle in radians whose sine value is the input. This function is the inverse of the sine function. The value can be a Decimal or Integer literal or a reference to a column containing numeric values. |
| ACOS Function | For input values between -1 and 1 inclusive, this function returns the angle in radians whose cosine value is the input. This function is the inverse of the cosine function. The value can be a Decimal or Integer literal or a reference to a column containing numeric values. |
| ATAN Function | For input values between -1 and 1 inclusive, this function returns the angle in radians whose tangent value is the input. This function is the inverse of the tangent function. The value can be a Decimal or Integer literal or a reference to a column containing numeric values. |

Also:

| Item | Description |
|---|---|
| ROUND Function | Rounds input value to the nearest integer. Input can be an Integer, a Decimal, a column reference, or an expression. Optional second argument can be used to specify the number of digits to which to round. |
| DEGREES Function | Computes the degrees of an input value measuring the radians of an angle. The value can be a Decimal or Integer literal or a reference to a column containing numeric values. |

> **NOTE:** These functions are valid over specific ranges.

The following functions are computed using the above functions.

- Arccotangent. Computed using ATAN function. See below.
- Arcsecant. Computed using ACOS function. See below.
- Arccosecant. Computed using ASIN function. See below.

**Source:**

In the following sample, input values are in radians. In this example, all values are rounded to two decimals for clarity.

| Y |
|---|
| -1.00 |
| -0.75 |
| -0.50 |
| 0.00 |
| 0.50 |
| 0.75 |
| 1.00 |

**Transformation:**

Arcsine:

Valid over the range (-1 <= Y <= 1)

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | round(degrees(asin(Y)), 2) |
| Parameter: New column name | 'asinY' |

Arccosine:

Valid over the range (-1 <= Y <= 1)

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | round(degrees(acos(Y)), 2) |
| Parameter: New column name | 'acosY' |

Arctangent:

Valid over the range (-1 <= Y <= 1)

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | round(degrees(atan(Y)), 2) |
| Parameter: New column name | 'atanY' |

Arccosecant:

This function is valid over a set of ranged inputs, so you can use a conditional column for the computation.

| Transformation Name | Conditional column |
|---|---|
| Parameter: Condition type | if...then...else |
| Parameter: If | (Y <= -1) \|\| (Y >= 1) |
| Parameter: Then | round(degrees(asin(divide(1, Y))), 2) |
| Parameter: New column name | 'acscY' |

Arcsecant:

Same set of ranged inputs apply to this function.

| Transformation Name | Conditional column |
|---|---|
| Parameter: Condition type | if...then...else |
| Parameter: If | (Y <= -1) \|\| (Y >= 1) |
| Parameter: Then | round(degrees(acos(divide(1, Y))), 2) |
| Parameter: New column name | 'asecY' |

Arccotangent:

For this function, the two different ranges of inputs have different computations, so an `else` condition is added to the transformation.

| Transformation Name | Conditional column |
|---|---|
| Parameter: Condition type | if...then...else |
| Parameter: If | Y > 0 |
| Parameter: Then | round(degrees(atan(divide(1, Y))), 2) |
| Parameter: Else | round(degrees(atan(divide(1, Y)) + pi()), 2) |
| Parameter: New column name | 'acotY' |

**Results:**

| Y | acotY | asecY | acscY | atanY | acosY | asinY |
|---|---|---|---|---|---|---|
| -1.00 | -41.86 | 180.00 | -90.00 | -45.00 | 180.00 | -90.00 |
| -0.75 | -49.99 | *null* | *null* | -37.00 | 139.00 | -49.00 |
| -0.50 | -60.29 | *null* | *null* | -27.00 | 120.00 | -30.00 |
| 0.00 | *null* | *null* | *null* | 0.00 | 90.00 | 0.00 |
| 0.50 | 63.44 | *null* | *null* | 27.00 | 60.00 | 30.00 |

| 0.75 | 53.13 | *null* | *null* | 37.00 | 41.00 | 49.00 |
| 1.00 | 45.00 | 0.00 | 90.00 | 45.00 | 0.00 | 90.00 |

# ATAN Function

For input values between -1 and 1 inclusive, this function returns the angle in radians whose tangent value is the input. This function is the inverse of the tangent function. The value can be a Decimal or Integer literal or a reference to a column containing numeric values.

> **NOTE:** While this function returns values outside of the range -1 <= x <= 1, those values are not considered valid.

For more information on the tangent function, see *TAN Function*.

**arc cotangent:**

| Input range | Output computation |
|---|---|
| x>0 | `atan(1/x)` |
| x<=0 | `atan(1/x) + PI()` |

**Wrangle vs. SQL:** This function is part of Wrangle , a proprietary data transformation language. Wrangle is not SQL. For more information, see *Wrangle Language*.

## Basic Usage

**Numeric literal example:**

```
atan(0.5)
```

**Output:** Returns the computation of the arc tangent of 0.5. Output value is in radians.

**Column reference example:**

```
atan(X)
```

**Output:** Returns the arc tangent of the values in X column.

## Syntax and Arguments

```
atan(numeric_value)
```

| Argument | Required? | Data Type | Description |
|---|---|---|---|
| numeric_val ue | Y | string, decimal, or integer | Name of column, Decimal or Integer literal, or function returning those types to apply to the function |

For more information on syntax standards, see *Language Documentation Syntax Notes*.

**numeric_value**

Name of the column, Integer or Decimal literal, or function returning that data type to apply to the function.

- Missing input values generate missing results.
- Literal numeric values should not be quoted. Quoted values are treated as strings.
- Multiple columns and wildcards are not supported.

**Usage Notes:**

| Required? | Data Type | Example Value |
|---|---|---|
| Yes | String (column reference) or Integer or Decimal literal | `0.5` |

## Examples

> **Tip:** For additional examples, see *Common Tasks*.

### Example - Trigonometry Arc functions

This example illustrates how to apply the inverse trigonometric (Arc) functions to your transformations.

**Functions:**

| Item | Description |
|---|---|
| ASIN Function | For input values between -1 and 1 inclusive, this function returns the angle in radians whose sine value is the input. This function is the inverse of the sine function. The value can be a Decimal or Integer literal or a reference to a column containing numeric values. |
| ACOS Function | For input values between -1 and 1 inclusive, this function returns the angle in radians whose cosine value is the input. This function is the inverse of the cosine function. The value can be a Decimal or Integer literal or a reference to a column containing numeric values. |
| ATAN Function | For input values between -1 and 1 inclusive, this function returns the angle in radians whose tangent value is the input. This function is the inverse of the tangent function. The value can be a Decimal or Integer literal or a reference to a column containing numeric values. |

Also:

| Item | Description |
|---|---|
| ROUND Function | Rounds input value to the nearest integer. Input can be an Integer, a Decimal, a column reference, or an expression. Optional second argument can be used to specify the number of digits to which to round. |
| DEGREES Function | Computes the degrees of an input value measuring the radians of an angle. The value can be a Decimal or Integer literal or a reference to a column containing numeric values. |

> **NOTE:** These functions are valid over specific ranges.

The following functions are computed using the above functions.

- Arccotangent. Computed using ATAN function. See below.
- Arcsecant. Computed using ACOS function. See below.
- Arccosecant. Computed using ASIN function. See below.

**Source:**

In the following sample, input values are in radians. In this example, all values are rounded to two decimals for clarity.

| Y |
|---|
| -1.00 |
| -0.75 |
| -0.50 |
| 0.00 |
| 0.50 |
| 0.75 |
| 1.00 |

**Transformation:**

Arcsine:

Valid over the range (-1 <= Y <= 1)

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | round(degrees(asin(Y)), 2) |
| Parameter: New column name | 'asinY' |

Arccosine:

Valid over the range (-1 <= Y <= 1)

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | round(degrees(acos(Y)), 2) |
| Parameter: New column name | 'acosY' |

Arctangent:

Valid over the range (-1 <= Y <= 1)

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | round(degrees(atan(Y)), 2) |
| Parameter: New column name | 'atanY' |

Arccosecant:

This function is valid over a set of ranged inputs, so you can use a conditional column for the computation.

| Transformation Name | Conditional column |
|---|---|
| Parameter: Condition type | `if...then...else` |
| Parameter: If | `(Y <= -1) || (Y >= 1)` |
| Parameter: Then | `round(degrees(asin(divide(1, Y))), 2)` |
| Parameter: New column name | `'acscY'` |

Arcsecant:

Same set of ranged inputs apply to this function.

| Transformation Name | Conditional column |
|---|---|
| Parameter: Condition type | `if...then...else` |
| Parameter: If | `(Y <= -1) || (Y >= 1)` |
| Parameter: Then | `round(degrees(acos(divide(1, Y))), 2)` |
| Parameter: New column name | `'asecY'` |

Arccotangent:

For this function, the two different ranges of inputs have different computations, so an `else` condition is added to the transformation.

| Transformation Name | Conditional column |
|---|---|
| Parameter: Condition type | `if...then...else` |
| Parameter: If | `Y > 0` |
| Parameter: Then | `round(degrees(atan(divide(1, Y))), 2)` |
| Parameter: Else | `round(degrees(atan(divide(1, Y)) + pi()), 2)` |
| Parameter: New column name | `'acotY'` |

**Results:**

| Y | acotY | asecY | acscY | atanY | acosY | asinY |
|---|---|---|---|---|---|---|
| -1.00 | -41.86 | 180.00 | -90.00 | -45.00 | 180.00 | -90.00 |
| -0.75 | -49.99 | *null* | *null* | -37.00 | 139.00 | -49.00 |
| -0.50 | -60.29 | *null* | *null* | -27.00 | 120.00 | -30.00 |
| 0.00 | *null* | *null* | *null* | 0.00 | 90.00 | 0.00 |
| 0.50 | 63.44 | *null* | *null* | 27.00 | 60.00 | 30.00 |
| 0.75 | 53.13 | *null* | *null* | 37.00 | 41.00 | 49.00 |
| 1.00 | 45.00 | 0.00 | 90.00 | 45.00 | 0.00 | 90.00 |

# SINH Function

Computes the hyperbolic sine of an input value for a hyperbolic angle measured in radians. The value can be a Decimal or Integer literal or a reference to a column containing numeric values.

A **hyperbola** is the shape created by taking a planar slice of two cones whose tips are touching each other. For two identical cones, the curves of the slices mirror each other, no matter the angle of the plane through the cones.

- The two slices represent the set of points on a grid such that:

$$x^2 - y^2 = k$$

  where `k` is some constant.
- The hyperbolic trigonometric functions measure trigonometric calculations for the right-side (x > 0) slice of the hyperbola.
- For more information, see *https://en.wikipedia.org/wiki/Hyperbola*.

The hyperbolic sine (SINH) function is computed using the following formula:

$$\sinh z = \frac{e^z - e^{-z}}{2}$$

**Hyperbolic cosecant:**

The hyperbolic cosecant is the following:

$$\operatorname{csch} z = \frac{1}{\sinh z}$$

You can convert from degrees to radians. For more information, see *RADIANS Function*.

**Wrangle vs. SQL:** This function is part of Wrangle , a proprietary data transformation language. Wrangle is not SQL. For more information, see *Wrangle Language*.

## Basic Usage

**Numeric literal example:**

```
round(sinh(radians(30)),3)
```

**Output:** Returns the computation of the hyperbolic sine of a 30-degree angle, which is converted to radians before being passed to the `SINH` function. The output value is rounded to three decimals: `0.548`.

**Column reference example:**

```
sinh(X)
```

**Output:** Returns the hyperbolic sine of the radians values in `X` column.

## Syntax and Arguments

```
sinh(numeric_value)
```

| Argument | Required? | Data Type | Description |
|---|---|---|---|
| numeric_value | Y | string, decimal, or integer | Name of column, Decimal or Integer literal, or function returning those types to apply to the function |

For more information on syntax standards, see *Language Documentation Syntax Notes*.

### numeric_value

Name of the column, Integer or Decimal literal, or function returning that data type to apply to the function.

- Missing input values generate missing results.
- Literal numeric values should not be quoted. Quoted values are treated as strings.
- Multiple columns and wildcards are not supported.

### Usage Notes:

| Required? | Data Type | Example Value |
|---|---|---|
| Yes | String (column reference) or Integer or Decimal literal | `0.5` |

## Examples

> **Tip:** For additional examples, see *Common Tasks*.

### Example - Hyperbolic trigonometry functions

This example illustrates how to apply hyperbolic trigonometric functions to your transformations. All of the functions take inputs in radians.

### Functions:

| Item | Description |
|---|---|
| SINH Function | Computes the hyperbolic sine of an input value for a hyperbolic angle measured in radians. The value can be a Decimal or Integer literal or a reference to a column containing numeric values. |
| COSH Function | Computes the hyperbolic cosine of an input value for a hyperbolic angle measured in radians. The value can be a Decimal or Integer literal or a reference to a column containing numeric values. |
| TANH Function | Computes the hyperbolic tangent of an input value for a hyperbolic angle measured in radians. The value can be a Decimal or Integer literal or a reference to a column containing numeric values. |

The following functions can be computed based on the inverse of the above functions:

- Hyperbolic Cotangent. Computed as 1/TANH.
- Hyperbolic Secant. Computed as 1/COSH.
- Hyperbolic Cosecant. Computed as 1/SINH.

Also:

| Item | Description |
|---|---|
| ROUND Function | Rounds input value to the nearest integer. Input can be an Integer, a Decimal, a column reference, or an expression. Optional second argument can be used to specify the number of digits to which to round. |
| RADIANS Function | Computes the radians of an input value measuring degrees of an angle. The value can be a Decimal or Integer literal or a reference to a column containing numeric values. |
| DIVIDE Function | Returns the value of dividing the first argument by the second argument. Equivalent to the / operator. |

**Source:**

In the following sample, input values are in degrees:

| X |
|---|
| -30 |
| 0 |
| 30 |
| 45 |
| 60 |
| 90 |
| 120 |
| 135 |
| 180 |

**Transformation:**

In this example, all values are rounded to three decimals for clarity.

First, the above values in degrees must be converted to radians.

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | round(radians(X), 3) |
| Parameter: New column name | 'rX' |

Hyperbolic Sine:

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | round(sinh(rX), 3) |
| Parameter: New column name | 'SINHrX' |

Hyperbolic Cosine:

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | round(cosh(rX), 3) |
| Parameter: New column name | 'COSHrX' |

Hyperbolic Tangent:

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | round(tanh(rX), 3) |
| Parameter: New column name | 'TANHrX' |

Hyperbolic Cotangent:

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | round(divide(1, tanh(rX)), 3) |
| Parameter: New column name | 'COTHrX' |

Hyperbolic Secant:

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | round(divide(1, cosh(rX)), 3) |
| Parameter: New column name | 'SECHrX' |

Hyperbolic Cosecant:

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | round(divide(1, sinh(rX)), 3) |
| Parameter: New column name | 'CSCHrX' |

**Results:**

| X | rX | TANHrX | COTHrX | COSHrX | SECHrX | SINHrX | CSCHrX |
|---|---|---|---|---|---|---|---|
| -30 | -0.524 | -0.481 | -2.079 | 1.14 | 0.877 | -0.548 | -1.825 |
| 0 | 0 | 0 | null | 1 | 1 | 0 | null |

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 30 | 0.524 | 0.481 | 2.079 | 1.14 | 0.877 | 0.548 | 1.825 |
| 45 | 0.785 | 0.656 | 1.524 | 1.324 | 0.755 | 0.868 | 1.152 |
| 60 | 1.047 | 0.781 | 1.28 | 1.6 | 0.625 | 1.249 | 0.801 |
| 90 | 1.571 | 0.917 | 1.091 | 2.51 | 0.398 | 2.302 | 0.434 |
| 120 | 2.094 | 0.97 | 1.031 | 4.12 | 0.243 | 3.997 | 0.25 |
| 135 | 2.356 | 0.982 | 1.018 | 5.322 | 0.188 | 5.227 | 0.191 |
| 180 | 3.142 | 0.996 | 1.004 | 11.597 | 0.086 | 11.553 | 0.087 |

# COSH Function

Computes the hyperbolic cosine of an input value for a hyperbolic angle measured in radians. The value can be a Decimal or Integer literal or a reference to a column containing numeric values.

A **hyperbola** is the shape created by taking a planar slice of two cones whose tips are touching each other. For two identical cones, the curves of the slices mirror each other, no matter the angle of the plane through the cones.

- The two slices represent the set of points on a grid such that:

$$x^2 - y^2 = k$$

  where `k` is some constant.
- The hyperbolic trigonometric functions measure trigonometric calculations for the right-side (x > 0) slice of the hyperbola.
- For more information, see *https://en.wikipedia.org/wiki/Hyperbola*.

The hyperbolic cosine (COSH) function is computed using the following formula:

$$\cosh z = \frac{e^z + e^{-z}}{2}$$

**Hyperbolic secant:**

The hyperbolic secant is the following:

$$\operatorname{sech} z = \frac{1}{\cosh z}$$

You can convert from degrees to radians. For more information, see *RADIANS Function*.

**Wrangle vs. SQL:** This function is part of Wrangle , a proprietary data transformation language. Wrangle is not SQL. For more information, see *Wrangle Language*.

## Basic Usage

**Numeric literal example:**

```
round(cosh(radians(30)),3)
```

**Output:** Returns the computation of the hyperbolic cosine of a 30-degree angle, which is converted to radians before being passed to the `COSH` function. The output value is rounded to three decimals: `1.140`.

**Column reference example:**

```
cosh(X)
```

**Output:** Returns the hyperbolic cosine of the radians values in `X` column.

## Syntax and Arguments

```
cosh(numeric_value)
```

| Argument | Required? | Data Type | Description |
|---|---|---|---|
| numeric_val ue | Y | string, decimal, or integer | Name of column, Decimal or Integer literal, or function returning those types to apply to the function |

For more information on syntax standards, see *Language Documentation Syntax Notes*.

### numeric_value

Name of the column, Integer or Decimal literal, or function returning that data type to apply to the function.

- Missing input values generate missing results.
- Literal numeric values should not be quoted. Quoted values are treated as strings.
- Multiple columns and wildcards are not supported.

**Usage Notes:**

| Required? | Data Type | Example Value |
|---|---|---|
| Yes | String (column reference) or Integer or Decimal literal | `0.5` |

## Examples

> **Tip:** For additional examples, see *Common Tasks*.

### Example - Hyperbolic trigonometry functions

This example illustrates how to apply hyperbolic trigonometric functions to your transformations. All of the functions take inputs in radians.

**Functions:**

| Item | Description |
|---|---|
| SINH Function | Computes the hyperbolic sine of an input value for a hyperbolic angle measured in radians. The value can be a Decimal or Integer literal or a reference to a column containing numeric values. |
| COSH Function | Computes the hyperbolic cosine of an input value for a hyperbolic angle measured in radians. The value can be a Decimal or Integer literal or a reference to a column containing numeric values. |
| TANH Function | Computes the hyperbolic tangent of an input value for a hyperbolic angle measured in radians. The value can be a Decimal or Integer literal or a reference to a column containing numeric values. |

The following functions can be computed based on the inverse of the above functions:

- Hyperbolic Cotangent. Computed as 1/TANH.
- Hyperbolic Secant. Computed as 1/COSH.
- Hyperbolic Cosecant. Computed as 1/SINH.

Also:

| Item | Description |
|------|-------------|
| ROUND Function | Rounds input value to the nearest integer. Input can be an Integer, a Decimal, a column reference, or an expression. Optional second argument can be used to specify the number of digits to which to round. |
| RADIANS Function | Computes the radians of an input value measuring degrees of an angle. The value can be a Decimal or Integer literal or a reference to a column containing numeric values. |
| DIVIDE Function | Returns the value of dividing the first argument by the second argument. Equivalent to the / operator. |

**Source:**

In the following sample, input values are in degrees:

| X |
|---|
| -30 |
| 0 |
| 30 |
| 45 |
| 60 |
| 90 |
| 120 |
| 135 |
| 180 |

**Transformation:**

In this example, all values are rounded to three decimals for clarity.

First, the above values in degrees must be converted to radians.

| Transformation Name | New formula |
|---------------------|-------------|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | round(radians(X), 3) |
| Parameter: New column name | 'rX' |

Hyperbolic Sine:

| Transformation Name | New formula |
|---------------------|-------------|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | round(sinh(rX), 3) |
| Parameter: New column name | 'SINHrX' |

Hyperbolic Cosine:

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | round(cosh(rX), 3) |
| Parameter: New column name | 'COSHrX' |

Hyperbolic Tangent:

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | round(tanh(rX), 3) |
| Parameter: New column name | 'TANHrX' |

Hyperbolic Cotangent:

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | round(divide(1, tanh(rX)), 3) |
| Parameter: New column name | 'COTHrX' |

Hyperbolic Secant:

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | round(divide(1, cosh(rX)), 3) |
| Parameter: New column name | 'SECHrX' |

Hyperbolic Cosecant:

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | round(divide(1, sinh(rX)), 3) |
| Parameter: New column name | 'CSCHrX' |

**Results:**

| X | rX | TANHrX | COTHrX | COSHrX | SECHrX | SINHrX | CSCHrX |
|---|---|---|---|---|---|---|---|
| -30 | -0.524 | -0.481 | -2.079 | 1.14 | 0.877 | -0.548 | -1.825 |
| 0 | 0 | 0 | *null* | 1 | 1 | 0 | *null* |

| 30 | 0.524 | 0.481 | 2.079 | 1.14 | 0.877 | 0.548 | 1.825 |
|----|-------|-------|-------|------|-------|-------|-------|
| 45 | 0.785 | 0.656 | 1.524 | 1.324 | 0.755 | 0.868 | 1.152 |
| 60 | 1.047 | 0.781 | 1.28 | 1.6 | 0.625 | 1.249 | 0.801 |
| 90 | 1.571 | 0.917 | 1.091 | 2.51 | 0.398 | 2.302 | 0.434 |
| 120 | 2.094 | 0.97 | 1.031 | 4.12 | 0.243 | 3.997 | 0.25 |
| 135 | 2.356 | 0.982 | 1.018 | 5.322 | 0.188 | 5.227 | 0.191 |
| 180 | 3.142 | 0.996 | 1.004 | 11.597 | 0.086 | 11.553 | 0.087 |

# TANH Function

Computes the hyperbolic tangent of an input value for a hyperbolic angle measured in radians. The value can be a Decimal or Integer literal or a reference to a column containing numeric values.

A **hyperbola** is the shape created by taking a planar slice of two cones whose tips are touching each other. For two identical cones, the curves of the slices mirror each other, no matter the angle of the plane through the cones.

- The two slices represent the set of points on a grid such that:

$$x^2 - y^2 = k$$

  where k is some constant.
- The hyperbolic trigonometric functions measure trigonometric calculations for the right-side (x > 0) slice of the hyperbola.
- For more information, see *https://en.wikipedia.org/wiki/Hyperbola*.

The hyperbolic tangent (TANH) function is computed using the following formula:

$$\tanh z = \frac{\sinh z}{\cosh z}$$

**Hyperbolic cotangent:**

The hyperbolic cotangent is the following:

$$\coth z = \frac{\cosh z}{\sinh z}$$

You can convert from degrees to radians. For more information, see *RADIANS Function*.

**Wrangle vs. SQL:** This function is part of Wrangle , a proprietary data transformation language. Wrangle is not SQL. For more information, see *Wrangle Language*.

## Basic Usage

**Numeric literal example:**

```
round(tanh(radians(30)),3)
```

**Output:** Returns the computation of the hyperbolic tangent of a 30-degree angle, which is converted to radians before being passed to the `TANH` function. The output value is rounded to three decimals: `0.548`.

**Column reference example:**

```
tanh(X)
```

**Output:** Returns the hyperbolic tangent of the radians values in `X` column.

## Syntax and Arguments

```
tanh(numeric_value)
```

| Argument | Required? | Data Type | Description |
|---|---|---|---|
| numeric_val ue | Y | string, decimal, or integer | Name of column, Decimal or Integer literal, or function returning those types to apply to the function |

For more information on syntax standards, see *Language Documentation Syntax Notes*.

### numeric_value

Name of the column, Integer or Decimal literal, or function returning that data type to apply to the function.

- Missing input values generate missing results.
- Literal numeric values should not be quoted. Quoted values are treated as strings.
- Multiple columns and wildcards are not supported.

**Usage Notes:**

| Required? | Data Type | Example Value |
|---|---|---|
| Yes | String (column reference) or Integer or Decimal literal | `0.5` |

## Examples

> **Tip:** For additional examples, see *Common Tasks.*

### Example - Hyperbolic trigonometry functions

This example illustrates how to apply hyperbolic trigonometric functions to your transformations. All of the functions take inputs in radians.

**Functions:**

| Item | Description |
|---|---|
| SINH Function | Computes the hyperbolic sine of an input value for a hyperbolic angle measured in radians. The value can be a Decimal or Integer literal or a reference to a column containing numeric values. |
| COSH Function | Computes the hyperbolic cosine of an input value for a hyperbolic angle measured in radians. The value can be a Decimal or Integer literal or a reference to a column containing numeric values. |
| TANH Function | Computes the hyperbolic tangent of an input value for a hyperbolic angle measured in radians. The value can be a Decimal or Integer literal or a reference to a column containing numeric values. |

The following functions can be computed based on the inverse of the above functions:

- Hyperbolic Cotangent. Computed as 1/TANH.
- Hyperbolic Secant. Computed as 1/COSH.
- Hyperbolic Cosecant. Computed as 1/SINH.

Also:

| Item | Description |
|------|-------------|
| ROUND Function | Rounds input value to the nearest integer. Input can be an Integer, a Decimal, a column reference, or an expression. Optional second argument can be used to specify the number of digits to which to round. |
| RADIANS Function | Computes the radians of an input value measuring degrees of an angle. The value can be a Decimal or Integer literal or a reference to a column containing numeric values. |
| DIVIDE Function | Returns the value of dividing the first argument by the second argument. Equivalent to the / operator. |

**Source:**

In the following sample, input values are in degrees:

| X |
|------|
| -30 |
| 0 |
| 30 |
| 45 |
| 60 |
| 90 |
| 120 |
| 135 |
| 180 |

**Transformation:**

In this example, all values are rounded to three decimals for clarity.

First, the above values in degrees must be converted to radians.

| Transformation Name | New formula |
|---------------------|-------------|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | round(radians(X), 3) |
| Parameter: New column name | 'rX' |

Hyperbolic Sine:

| Transformation Name | New formula |
|---------------------|-------------|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | round(sinh(rX), 3) |
| Parameter: New column name | 'SINHrX' |

Hyperbolic Cosine:

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | round(cosh(rX), 3) |
| Parameter: New column name | 'COSHrX' |

Hyperbolic Tangent:

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | round(tanh(rX), 3) |
| Parameter: New column name | 'TANHrX' |

Hyperbolic Cotangent:

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | round(divide(1, tanh(rX)), 3) |
| Parameter: New column name | 'COTHrX' |

Hyperbolic Secant:

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | round(divide(1, cosh(rX)), 3) |
| Parameter: New column name | 'SECHrX' |

Hyperbolic Cosecant:

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | round(divide(1, sinh(rX)), 3) |
| Parameter: New column name | 'CSCHrX' |

**Results:**

| X | rX | TANHrX | COTHrX | COSHrX | SECHrX | SINHrX | CSCHrX |
|---|---|---|---|---|---|---|---|
| -30 | -0.524 | -0.481 | -2.079 | 1.14 | 0.877 | -0.548 | -1.825 |
| 0 | 0 | 0 | *null* | 1 | 1 | 0 | *null* |

| 30 | 0.524 | 0.481 | 2.079 | 1.14 | 0.877 | 0.548 | 1.825 |
| 45 | 0.785 | 0.656 | 1.524 | 1.324 | 0.755 | 0.868 | 1.152 |
| 60 | 1.047 | 0.781 | 1.28 | 1.6 | 0.625 | 1.249 | 0.801 |
| 90 | 1.571 | 0.917 | 1.091 | 2.51 | 0.398 | 2.302 | 0.434 |
| 120 | 2.094 | 0.97 | 1.031 | 4.12 | 0.243 | 3.997 | 0.25 |
| 135 | 2.356 | 0.982 | 1.018 | 5.322 | 0.188 | 5.227 | 0.191 |
| 180 | 3.142 | 0.996 | 1.004 | 11.597 | 0.086 | 11.553 | 0.087 |

# ASINH Function

Computes the arcsine of an input value for a hyperbolic angle measured in radians. The value can be a Decimal or Integer literal or a reference to a column containing numeric values.

A **hyperbola** is the shape created by taking a planar slice of two cones whose tips are touching each other. For two identical cones, the curves of the slices mirror each other, no matter the angle of the plane through the cones.

- The two slices represent the set of points on a grid such that:

$$x^2 - y^2 = k$$

  where k is some constant.
- The hyperbolic trigonometric functions measure trigonometric calculations for the right-side (x > 0) slice of the hyperbola.
- For more information, see *https://en.wikipedia.org/wiki/Hyperbola*.

The hyperbolic arcsine (ASINH) function is computed using the following formula:

$$\sinh^{-1} z = \ln(z + \sqrt{z^2 + 1})$$

You can convert from degrees to radians. For more information, see *RADIANS Function*.

**Wrangle vs. SQL:** This function is part of Wrangle , a proprietary data transformation language. Wrangle is not SQL. For more information, see *Wrangle Language*.

## Basic Usage

**Numeric literal example:**

```
round(asinh(0.5),3)
```

**Output:** Returns the computation of the hyperbolic angle in radians whose sine is 0.5. The output value is rounded to three decimals: 0.481.

**Column reference example:**

```
asinh(X)
```

**Output:** Returns the hyperbolic angle the sine value for which is stored in radians in X column.

## Syntax and Arguments

```
asinh(numeric_value)
```

| Argument | Required? | Data Type | Description |
|---|---|---|---|
| numeric_value | Y | string, decimal, or integer | Name of column, Decimal or Integer literal, or function returning those types to apply to the function |

For more information on syntax standards, see *Language Documentation Syntax Notes*.

**numeric_value**

Name of the column, Integer or Decimal literal, or function returning that data type to apply to the function.

- Missing input values generate missing results.
- Literal numeric values should not be quoted. Quoted values are treated as strings.
- Multiple columns and wildcards are not supported.

**Usage Notes:**

| Required? | Data Type | Example Value |
|---|---|---|
| Yes | String (column reference) or Integer or Decimal literal | `0.5` |

## Examples

> **Tip:** For additional examples, see *Common Tasks*.

**Example - Hyperbolic arc trigonometry functions**

This example illustrates how to apply inverse (arc) hyperbolic functions to your transformations.

**Functions:**

| Item | Description |
|---|---|
| ASINH Function | Computes the arcsine of an input value for a hyperbolic angle measured in radians. The value can be a Decimal or Integer literal or a reference to a column containing numeric values. |
| ACOSH Function | Computes the arccosine of an input value for a hyperbolic angle measured in radians. The value can be a Decimal or Integer literal or a reference to a column containing numeric values. |
| ATANH Function | Computes the arctangent of an input value for a hyperbolic angle measured in radians. The value can be a Decimal or Integer literal or a reference to a column containing numeric values. |

Also:

| Item | Description |
|---|---|
| ROUND Function | Rounds input value to the nearest integer. Input can be an Integer, a Decimal, a column reference, or an expression. Optional second argument can be used to specify the number of digits to which to round. |
| DEGREES Function | Computes the degrees of an input value measuring the radians of an angle. The value can be a Decimal or Integer literal or a reference to a column containing numeric values. |
| NULL Function | The `NULL` function generates null values. |

**Source:**

In the following sample, input values are in radians. In this example, all values are rounded to two decimals for clarity.

| Y |
|---|

| |
|---|
| -4.00 |
| -3.00 |
| -2.00 |
| -1.00 |
| -0.75 |
| -0.50 |
| 0.00 |
| 0.50 |
| 0.75 |
| 1.00 |
| 2.00 |
| 3.00 |
| 4.00 |

**Transformation:**

The following transformations include checks for the valid ranges for input values.

Hyperbolic arcsine:

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | round(degrees(asinh(Y)), 2) |
| Parameter: New column name | 'asinhY' |

Hyperbolic arccosine:

Valid over the range (y > 1)

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | if(Y>1,round(degrees(acosh(Y)), 2),null()) |
| Parameter: New column name | 'acoshY' |

Hyperbolic arctangent:

Valid over the range (-1 < y < 1)

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | if(abs(y)<1,round(degrees(atanh(Y)), 2),null()) |

| Parameter: New column name | 'atanhY' |
| --- | --- |

**Results:**

| Y | atanhY | acoshY | asinhY |
| --- | --- | --- | --- |
| -4 | *null* | *null* | -120.02 |
| -3 | *null* | *null* | -104.19 |
| -2 | *null* | *null* | -82.71 |
| -1.5 | *null* | *null* | -68.45 |
| -1 | *null* | *null* | -50.5 |
| -0.75 | -55.75 | *null* | -39.71 |
| -0.5 | -31.47 | *null* | -27.57 |
| 0 | 0 | *null* | 0 |
| 0.5 | 31.47 | *null* | 27.57 |
| 0.75 | 55.75 | *null* | 39.71 |
| 1 | *null* | *null* | 50.5 |
| 1.5 | *null* | 55.14 | 68.45 |
| 2 | *null* | 75.46 | 82.71 |
| 3 | *null* | 101 | 104.19 |
| 4 | *null* | 118.23 | 120.02 |

# ACOSH Function

Computes the arccosine of an input value for a hyperbolic angle measured in radians. The value can be a Decimal or Integer literal or a reference to a column containing numeric values.

A **hyperbola** is the shape created by taking a planar slice of two cones whose tips are touching each other. For two identical cones, the curves of the slices mirror each other, no matter the angle of the plane through the cones.

- The two slices represent the set of points on a grid such that:

$$x^2 - y^2 = k$$

  where $k$ is some constant.
- The hyperbolic trigonometric functions measure trigonometric calculations for the right-side (x > 0) slice of the hyperbola.
- For more information, see *https://en.wikipedia.org/wiki/Hyperbola*.

The hyperbolic arccosine (ACOSH) function is computed using the following formula:

$$\cosh^{-1} z = \ln(z + \sqrt{z^2 - 1})$$

You can convert from degrees to radians. For more information, see *RADIANS Function*.

**Wrangle vs. SQL:** This function is part of Wrangle , a proprietary data transformation language. Wrangle is not SQL. For more information, see *Wrangle Language*.

## Basic Usage

**Numeric literal example:**

```
round(degrees(acosh(1.5)), 2)
```

**Output:** Returns the computation of the hyperbolic angle in degrees whose cosine is `1.5`. The output value is rounded to two decimals: `55.14`.

**Column reference example:**

```
acosh(X)
```

**Output:** Returns the hyperbolic angle the cosine value for which is stored in radians in `X` column.

## Syntax and Arguments

```
acosh(numeric_value)
```

| Argument | Required? | Data Type | Description |
|---|---|---|---|
| numeric_value | Y | string, decimal, or integer | Name of column, Decimal or Integer literal, or function returning those types to apply to the function |

For more information on syntax standards, see *Language Documentation Syntax Notes*.

**numeric_value**

Name of the column, Integer or Decimal literal, or function returning that data type to apply to the function.

- Missing input values generate missing results.
- Literal numeric values should not be quoted. Quoted values are treated as strings.
- Multiple columns and wildcards are not supported.

**Usage Notes:**

| Required? | Data Type | Example Value |
|-----------|-----------|---------------|
| Yes | String (column reference) or Integer or Decimal literal | `0.5` |

## Examples

> **Tip:** For additional examples, see *Common Tasks*.

### Example - Hyperbolic arc trigonometry functions

This example illustrates how to apply inverse (arc) hyperbolic functions to your transformations.

**Functions:**

| Item | Description |
|------|-------------|
| ASINH Function | Computes the arcsine of an input value for a hyperbolic angle measured in radians. The value can be a Decimal or Integer literal or a reference to a column containing numeric values. |
| ACOSH Function | Computes the arccosine of an input value for a hyperbolic angle measured in radians. The value can be a Decimal or Integer literal or a reference to a column containing numeric values. |
| ATANH Function | Computes the arctangent of an input value for a hyperbolic angle measured in radians. The value can be a Decimal or Integer literal or a reference to a column containing numeric values. |

Also:

| Item | Description |
|------|-------------|
| ROUND Function | Rounds input value to the nearest integer. Input can be an Integer, a Decimal, a column reference, or an expression. Optional second argument can be used to specify the number of digits to which to round. |
| DEGREES Function | Computes the degrees of an input value measuring the radians of an angle. The value can be a Decimal or Integer literal or a reference to a column containing numeric values. |
| NULL Function | The `NULL` function generates null values. |

**Source:**

In the following sample, input values are in radians. In this example, all values are rounded to two decimals for clarity.

| Y |
|---|

| |
|---|
| -4.00 |
| -3.00 |
| -2.00 |
| -1.00 |
| -0.75 |
| -0.50 |
| 0.00 |
| 0.50 |
| 0.75 |
| 1.00 |
| 2.00 |
| 3.00 |
| 4.00 |

**Transformation:**

The following transformations include checks for the valid ranges for input values.

Hyperbolic arcsine:

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | round(degrees(asinh(Y)), 2) |
| Parameter: New column name | 'asinhY' |

Hyperbolic arccosine:

Valid over the range (y > 1)

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | if(Y>1,round(degrees(acosh(Y)), 2),null()) |
| Parameter: New column name | 'acoshY' |

Hyperbolic arctangent:

Valid over the range (-1 < y < 1)

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | if(abs(y)<1,round(degrees(atanh(Y)), 2),null()) |

| Parameter: New column name | 'atanhY' |
|---|---|

**Results:**

| Y | atanhY | acoshY | asinhY |
|---|---|---|---|
| -4 | *null* | *null* | -120.02 |
| -3 | *null* | *null* | -104.19 |
| -2 | *null* | *null* | -82.71 |
| -1.5 | *null* | *null* | -68.45 |
| -1 | *null* | *null* | -50.5 |
| -0.75 | -55.75 | *null* | -39.71 |
| -0.5 | -31.47 | *null* | -27.57 |
| 0 | 0 | *null* | 0 |
| 0.5 | 31.47 | *null* | 27.57 |
| 0.75 | 55.75 | *null* | 39.71 |
| 1 | *null* | *null* | 50.5 |
| 1.5 | *null* | 55.14 | 68.45 |
| 2 | *null* | 75.46 | 82.71 |
| 3 | *null* | 101 | 104.19 |
| 4 | *null* | 118.23 | 120.02 |

# ATANH Function

Computes the arctangent of an input value for a hyperbolic angle measured in radians. The value can be a Decimal or Integer literal or a reference to a column containing numeric values.

A **hyperbola** is the shape created by taking a planar slice of two cones whose tips are touching each other. For two identical cones, the curves of the slices mirror each other, no matter the angle of the plane through the cones.

- The two slices represent the set of points on a grid such that:

$$x^2 - y^2 = k$$

where $k$ is some constant.
- The hyperbolic trigonometric functions measure trigonometric calculations for the right-side (x > 0) slice of the hyperbola.
- For more information, see *https://en.wikipedia.org/wiki/Hyperbola*.

The hyperbolic arctangent (ATANH) function is computed using the following formula:

$$\tanh^{-1} z = \frac{1}{2} \ln\left(\frac{1 + z}{1 - z}\right)$$

You can convert from degrees to radians. For more information, see *RADIANS Function*.

**Wrangle vs. SQL:** This function is part of Wrangle , a proprietary data transformation language. Wrangle is not SQL. For more information, see *Wrangle Language*.

## Basic Usage

**Numeric literal example:**

```
round(degrees(atanh(0.5)), 2)
```

**Output:** Returns the computation of the hyperbolic angle in radians whose tangent is `0.5`. The output value is rounded to two decimals: `31.47`.

**Column reference example:**

```
atanh(X)
```

**Output:** Returns the hyperbolic angle the tangent value for which is stored in radians in `X` column.

## Syntax and Arguments

```
atanh(numeric_value)
```

| Argument | Required? | Data Type | Description |
|----------|-----------|-----------|-------------|

| numeric_val ue | Y | string, decimal, or integer | Name of column, Decimal or Integer literal, or function returning those types to apply to the function |
|---|---|---|---|

For more information on syntax standards, see *Language Documentation Syntax Notes*.

### numeric_value

Name of the column, Integer or Decimal literal, or function returning that data type to apply to the function.

- Missing input values generate missing results.
- Literal numeric values should not be quoted. Quoted values are treated as strings.
- Multiple columns and wildcards are not supported.

### Usage Notes:

| Required? | Data Type | Example Value |
|---|---|---|
| Yes | String (column reference) or Integer or Decimal literal | `0.5` |

## Examples

> **Tip:** For additional examples, see *Common Tasks*.

### Example - Hyperbolic arc trigonometry functions

This example illustrates how to apply inverse (arc) hyperbolic functions to your transformations.

### Functions:

| Item | Description |
|---|---|
| ASINH Function | Computes the arcsine of an input value for a hyperbolic angle measured in radians. The value can be a Decimal or Integer literal or a reference to a column containing numeric values. |
| ACOSH Function | Computes the arccosine of an input value for a hyperbolic angle measured in radians. The value can be a Decimal or Integer literal or a reference to a column containing numeric values. |
| ATANH Function | Computes the arctangent of an input value for a hyperbolic angle measured in radians. The value can be a Decimal or Integer literal or a reference to a column containing numeric values. |

Also:

| Item | Description |
|---|---|
| ROUND Function | Rounds input value to the nearest integer. Input can be an Integer, a Decimal, a column reference, or an expression. Optional second argument can be used to specify the number of digits to which to round. |
| DEGREES Function | Computes the degrees of an input value measuring the radians of an angle. The value can be a Decimal or Integer literal or a reference to a column containing numeric values. |
| NULL Function | The `NULL` function generates null values. |

### Source:

In the following sample, input values are in radians. In this example, all values are rounded to two decimals for clarity.

| Y |
|---|
| -4.00 |
| -3.00 |
| -2.00 |
| -1.00 |
| -0.75 |
| -0.50 |
| 0.00 |
| 0.50 |
| 0.75 |
| 1.00 |
| 2.00 |
| 3.00 |
| 4.00 |

**Transformation:**

The following transformations include checks for the valid ranges for input values.

Hyperbolic arcsine:

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | round(degrees(asinh(Y)), 2) |
| Parameter: New column name | 'asinhY' |

Hyperbolic arccosine:

Valid over the range (y > 1)

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | if(Y>1,round(degrees(acosh(Y)), 2),null()) |
| Parameter: New column name | 'acoshY' |

Hyperbolic arctangent:

Valid over the range (-1 < y < 1)

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |

| Parameter: Formula | `if(abs(y)<1,round(degrees(atanh(Y)), 2),null())` |
|---|---|
| Parameter: New column name | `'atanhY'` |

**Results:**

| Y | atanhY | acoshY | asinhY |
|---|---|---|---|
| -4 | *null* | *null* | -120.02 |
| -3 | *null* | *null* | -104.19 |
| -2 | *null* | *null* | -82.71 |
| -1.5 | *null* | *null* | -68.45 |
| -1 | *null* | *null* | -50.5 |
| -0.75 | -55.75 | *null* | -39.71 |
| -0.5 | -31.47 | *null* | -27.57 |
| 0 | 0 | *null* | 0 |
| 0.5 | 31.47 | *null* | 27.57 |
| 0.75 | 55.75 | *null* | 39.71 |
| 1 | *null* | *null* | 50.5 |
| 1.5 | *null* | 55.14 | 68.45 |
| 2 | *null* | 75.46 | 82.71 |
| 3 | *null* | 101 | 104.19 |
| 4 | *null* | 118.23 | 120.02 |

# DEGREES Function

Computes the degrees of an input value measuring the radians of an angle. The value can be a Decimal or Integer literal or a reference to a column containing numeric values.

- Input units are in radians.
- You can convert from degrees to radians. For more information, see *RADIANS Function*.

**Wrangle vs. SQL:** This function is part of Wrangle , a proprietary data transformation language. Wrangle is not SQL. For more information, see *Wrangle Language*.

## Basic Usage

**Numeric literal example:**

```
round(degrees(1.0000),4)
```

**Output:** Returns the computation in degrees of `1.0000` radians, which is `57.2728`, rounded to 4 decimals.

**Column reference example:**

```
degrees(myRads)
```

**Output:** Returns the conversion of the values in `MyRads` column to degrees.

## Syntax and Arguments

```
degrees(numeric_value)
```

| Argument | Required? | Data Type | Description |
|---|---|---|---|
| numeric_val ue | Y | string, decimal, or integer | Name of column, Decimal or Integer literal, or function returning those types to apply to the function |

For more information on syntax standards, see *Language Documentation Syntax Notes*.

**numeric_value**

Name of the column, Integer or Decimal literal, or function returning that data type to apply to the function.

- Missing input values generate missing results.
- Literal numeric values should not be quoted. Quoted values are treated as strings.
- Multiple columns and wildcards are not supported.

**Usage Notes:**

| Required? | Data Type | Example Value |
|---|---|---|
| Yes | String (column reference) or Integer or Decimal literal | 3.14 |

## Examples

### Example - DEGREES and RADIANS functions

This example illustrates to convert values from one unit of measure to the other.

**Functions:**

| Item | Description |
|---|---|
| DEGREES Function | Computes the degrees of an input value measuring the radians of an angle. The value can be a Decimal or Integer literal or a reference to a column containing numeric values. |
| RADIANS Function | Computes the radians of an input value measuring degrees of an angle. The value can be a Decimal or Integer literal or a reference to a column containing numeric values. |
| ROUND Function | Rounds input value to the nearest integer. Input can be an Integer, a Decimal, a column reference, or an expression. Optional second argument can be used to specify the number of digits to which to round. |

**Source:**

In this example, the source data contains information about a set of isosceles triangles. Each triangle is listed in a separate row, with the listed value as the size of the non-congruent angle in the triangle in degrees.

You must calculate the measurement of all three angles of each isosceles triangle in radians.

| triangle | a01 |
|---|---|
| t01 | 30 |
| t02 | 60 |
| t03 | 90 |
| t04 | 120 |
| t05 | 150 |

**Transformation:**

You can convert the value for the non-congruent angle to radians using the following:

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | ROUND(RADIANS(a01), 4) |
| Parameter: New column name | 'r01' |

Now, calculate the value in degrees of the remaining two angles, which are congruent. Since the sum of all angles in a triangle is 180, the following formula can be applied to compute the size in degrees of each of these angles:

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |

| Parameter: Formula | (180 - a01) / 2 |
|---|---|
| Parameter: New column name | 'a02' |

Convert the above to radians:

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | ROUND(RADIANS(a02), 4) |
| Parameter: New column name | 'r02' |

Create a second column for the other congruent angle:

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | ROUND(RADIANS(a02), 4) |
| Parameter: New column name | 'r03' |

To check accuracy, you sum all three columns and convert to degrees:

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | ROUND(RADIANS(a02), 4) |
| Parameter: New column name | 'checksum' |

**Results:**

After you delete the intermediate columns, you see the following results and determine the error in the checksum is acceptable:

| triangle | a01 | r03 | r02 | r01 | checksum |
|---|---|---|---|---|---|
| t01 | 30 | 1.3095 | 1.3095 | 0.5238 | 179.9967 |
| t02 | 60 | 1.0476 | 1.0476 | 1.0476 | 179.9967 |
| t03 | 90 | 0.7857 | 0.7857 | 1.5714 | 179.9967 |
| t04 | 120 | 0.5238 | 0.5238 | 2.0952 | 179.9967 |
| t05 | 150 | 0.2619 | 0.2619 | 2.6190 | 179.9967 |

# RADIANS Function

Computes the radians of an input value measuring degrees of an angle. The value can be a Decimal or Integer literal or a reference to a column containing numeric values.

- A unit of 1 **radian** identifies the angle of a circle where the radius of the circle equals the length of the arc on the circle for that angle. This value corresponds to approximately 57.3 degrees.
- Input units are in degrees.
- You can convert from radians to degrees. For more information, see *DEGREES Function*.

**Wrangle vs. SQL:** This function is part of Wrangle , a proprietary data transformation language. Wrangle is not SQL. For more information, see *Wrangle Language*.

## Basic Usage

**Numeric literal example:**

```
round(radians(57.2728),4)
```

**Output:** Returns the computation in radians of `57.2728` rounded to four digits, which is `1.0000`.

**Column reference example:**

```
radians(myDegrees)
```

**Output:** Generates the new `myRads` column containing the conversion of the values in `MyDegrees` column to radians.

## Syntax and Arguments

```
radians(numeric_value)
```

| Argument | Required? | Data Type | Description |
|----------|-----------|-----------|-------------|
| numeric_value | Y | string, decimal, or integer | Name of column, Decimal or Integer literal, or function returning those types to apply to the function |

For more information on syntax standards, see *Language Documentation Syntax Notes*.

**numeric_value**

Name of the column, Integer or Decimal literal, or function returning that data type to apply to the function.

- Missing input values generate missing results.
- Literal numeric values should not be quoted. Quoted values are treated as strings.
- Multiple columns and wildcards are not supported.

**Usage Notes:**

| Required? | Data Type | Example Value |
|-----------|-----------|---------------|
| Yes | String (column reference) or Integer or Decimal literal | 10 |

# Examples

> **Tip:** For additional examples, see *Common Tasks*.

### Example - DEGREES and RADIANS functions

This example illustrates to convert values from one unit of measure to the other.

**Functions:**

| Item | Description |
|------|-------------|
| DEGREES Function | Computes the degrees of an input value measuring the radians of an angle. The value can be a Decimal or Integer literal or a reference to a column containing numeric values. |
| RADIANS Function | Computes the radians of an input value measuring degrees of an angle. The value can be a Decimal or Integer literal or a reference to a column containing numeric values. |
| ROUND Function | Rounds input value to the nearest integer. Input can be an Integer, a Decimal, a column reference, or an expression. Optional second argument can be used to specify the number of digits to which to round. |

**Source:**

In this example, the source data contains information about a set of isosceles triangles. Each triangle is listed in a separate row, with the listed value as the size of the non-congruent angle in the triangle in degrees.

You must calculate the measurement of all three angles of each isosceles triangle in radians.

| triangle | a01 |
|----------|-----|
| t01 | 30 |
| t02 | 60 |
| t03 | 90 |
| t04 | 120 |
| t05 | 150 |

**Transformation:**

You can convert the value for the non-congruent angle to radians using the following:

| Transformation Name | New formula |
|---------------------|-------------|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | ROUND(RADIANS(a01), 4) |
| Parameter: New column name | 'r01' |

Now, calculate the value in degrees of the remaining two angles, which are congruent. Since the sum of all angles in a triangle is 180, the following formula can be applied to compute the size in degrees of each of these angles:

| Transformation Name | New formula |
|---------------------|-------------|

| | |
|---|---|
| **Parameter: Formula type** | `Single row formula` |
| **Parameter: Formula** | `(180 - a01) / 2` |
| **Parameter: New column name** | `'a02'` |

Convert the above to radians:

| | |
|---|---|
| **Transformation Name** | `New formula` |
| **Parameter: Formula type** | `Single row formula` |
| **Parameter: Formula** | `ROUND(RADIANS(a02), 4)` |
| **Parameter: New column name** | `'r02'` |

Create a second column for the other congruent angle:

| | |
|---|---|
| **Transformation Name** | `New formula` |
| **Parameter: Formula type** | `Single row formula` |
| **Parameter: Formula** | `ROUND(RADIANS(a02), 4)` |
| **Parameter: New column name** | `'r03'` |

To check accuracy, you sum all three columns and convert to degrees:

| | |
|---|---|
| **Transformation Name** | `New formula` |
| **Parameter: Formula type** | `Single row formula` |
| **Parameter: Formula** | `ROUND(RADIANS(a02), 4)` |
| **Parameter: New column name** | `'checksum'` |

**Results:**

After you delete the intermediate columns, you see the following results and determine the error in the checksum is acceptable:

| triangle | a01 | r03 | r02 | r01 | checksum |
|---|---|---|---|---|---|
| t01 | 30 | 1.3095 | 1.3095 | 0.5238 | 179.9967 |
| t02 | 60 | 1.0476 | 1.0476 | 1.0476 | 179.9967 |
| t03 | 90 | 0.7857 | 0.7857 | 1.5714 | 179.9967 |
| t04 | 120 | 0.5238 | 0.5238 | 2.0952 | 179.9967 |
| t05 | 150 | 0.2619 | 0.2619 | 2.6190 | 179.9967 |

# Date Functions

Designer Cloud Powered by Trifacta® Enterprise Edition can process date information in a variety of sources. When date values are originally imported, they might be typed as String values due to inconsistencies in how they are represented. Using the functions below, you can set the formatting of your date values, so that they are consistently structured throughout your dataset.

Additionally, some of these functions enable you to extract numeric values from dates or to perform comparisons.

You can also convert date values between Unix time values and standard time values, which can be expressed to a precision of milliseconds.

> **NOTE:** Other than the `UNIXTIME` function, date functions ignore time zone in Datetime values.

**Date ranges:**

Valid dates fit in the following range: `01/01/1400` to `12/31/2599`.

# DATE Function

**Contents:**

---

Generates a date value from three inputs of Integer type: year, month, and day.

- Source values can be Integer literals or column references to values that can be inferred as Integers.
- If any of the source values are invalid or out of range, a missing value is returned.
- This function must be nested within another function that accepts date values, such as `DATEFORMAT`, as arguments. See the example below.

**Wrangle vs. SQL:** This function is part of Wrangle , a proprietary data transformation language. Wrangle is not SQL. For more information, see *Wrangle Language*.

## Basic Usage

**Integer literal values:**

```
dateformat(date(2015,02,15),&apos;yyyy-MM-dd&apos;)
```

**Output:** Returns the formatted date value: `2015-02-15`.

**Column reference values:**

```
dateformat(date(myYear, myMonth, myDay),&apos;MMM yyyy&apos;)
```

**Output:** Returns date values based on three columns, formatted for date.

## Syntax and Arguments

```
dateformat(date(year_integer_col,month_integer_col,day_Integer_col ),&apos;
date_format_string&apos;)
```

| Argument | Required? | Data Type | Description |
|---|---|---|---|
| year_integer_col | Y | integer | Name of column or Integer literal representing the year value to apply to the function |
| month_integer_col | Y | integer | Name of column or Integer literal representing the month value to apply to the function |
| day_integer_col | Y | integer | Name of column or Integer literal representing the day value to apply to the function |
| date_format_string | Y | string | String literal identifying the date format to apply to the value |

For more information on syntax standards, see *Language Documentation Syntax Notes*.

**year_integer_col**

Integer literal or name of the column containing integer values for the year.

- Missing values for this function in the source data result in missing values in the output.
- Multiple columns and wildcards are not supported.

**Usage Notes:**

| Required? | Data Type | Example Value |
|---|---|---|
| Yes | Integer (literal or column reference) | 2015 |

**month_integer_col**

Integer literal or name of the column containing integer values for the month.

- Values must be 1 or more, with a maximum value of 12.
- Missing values for this function in the source data result in missing values in the output.
- Multiple columns and wildcards are not supported.

**Usage Notes:**

| Required? | Data Type | Example Value |
|---|---|---|
| Yes | Integer (literal or column reference) | 4 |

**day_integer_col**

Integer literal or name of the column containing integer values for the day.

- Values must be 1 or more, with a maximum value for any month of 31.
- Missing values for this function in the source data result in missing values in the output.
- Multiple columns and wildcards are not supported.

**Usage Notes:**

| Required? | Data Type | Example Value |
|---|---|---|
| Yes | Integer (literal or column reference) | 26 |

**date_format_string**

String literal that describes how the date value is formatted.

**Usage Notes:**

| Required? | Data Type | Example Value |
|---|---|---|

| | | |
|---|---|---|
| Yes | String literal value | `'yyyy-MM-dd'` |

For more information, see *DATEFORMAT Function*.

## Examples

> **Tip:** For additional examples, see *Common Tasks*.

### Example - date and time functions

This example provides an overview on various date and time functions.

**Functions**:

| Item | Description |
|---|---|
| DATE Function | Generates a date value from three inputs of Integer type: year, month, and day. |
| TIME Function | Generates time values from three inputs of Integer type: hour, minute, and second. |
| DATETIME Function | Generates a Datetime value from the following inputs of Integer type: year, month, day, hour, minute, and second. |
| DATEFORMAT Function | Formats a specified Datetime set of values according to the specified date format. Source values can be a reference to a column containing Datetime values. |

**Source:**

| year | month | day | hour | minute | second |
|---|---|---|---|---|---|
| 2016 | 10 | 11 | 2 | 3 | 0 |
| 2015 | 11 | 20 | 15 | 22 | 30 |
| 2014 | 12 | 25 | 18 | 30 | 45 |

**Transformation:**

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | `DATEFORMAT(DATE (year, month, day),'yyyy-MM-dd')` |
| Parameter: New column name | `'fctn_date'` |

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | `DATEFORMAT(TIME (hour, minute, second),'HH-mm-ss')` |
| Parameter: New column name | `'fctn_time'` |

| Transformation Name | New formula |
|---|---|

| Parameter: Formula type | Single row formula |
|---|---|
| Parameter: Formula | DATEFORMAT(DATETIME (year, month, day, hour, minute, second),'yyyy-MM-dd-HH:mm:ss') |
| Parameter: New column name | 'fctn_datetime' |

**Results:**

> **NOTE:** All inputs must be inferred as Integer type and must be valid values for the specified input. For example, month values must be integers between 1 and 12, inclusive.

| year | month | day | hour | minute | second | fctn_date | fctn_time | fctn_datetime |
|---|---|---|---|---|---|---|---|---|
| 2016 | 10 | 11 | 2 | 3 | 0 | 2016-10-11 | 02-03-00 | 2016-10-11-02:03:00 |
| 2015 | 11 | 20 | 15 | 22 | 30 | 2015-11-20 | 15-22-30 | 2015-11-20-15:22:30 |
| 2014 | 12 | 25 | 18 | 30 | 45 | 2014-12-25 | 18-30-45 | 2014-12-25-18:30:45 |

You can apply other date and time functions to the generated columns. For an example, see *YEAR Function*.

# TIME Function

**Contents:**

---

Generates time values from three inputs of Integer type: hour, minute, and second.

- Source values can be Integer literals or column references to values that can be inferred as Integers.
- If any of the source values are invalid or out of range, a missing value is returned.
- This function must be nested within another function that accepts date values as arguments. See the example below.

**Wrangle vs. SQL:** This function is part of Wrangle , a proprietary data transformation language. Wrangle is not SQL. For more information, see *Wrangle Language*.

## Basic Usage

**Integer literal values:**

```
dateformat(time(23,58,59),&apos;HH mm ss&apos;)
```

**Output:** Returns the following map:

```
23 58 59
```

**Column reference values:**

```
dateformat(time(myHour, myMinute, mySecond), &apos;hh-mm-ss&apos;)
```

**Output:** Generates a column of values where:

- `hh` = values from `myHour` column
- `mm` = values from `myMinute` column
- `ss` = values from `mySecond` column

## Syntax and Arguments

```
dateformat(time(hour_integer_col,minute_integer_col,second_integer_col ),&apos;
time_format_string&apos;)
```

| Argument | Required? | Data Type | Description |
|---|---|---|---|
| hour_integer_col | Y | integer | Name of column or Integer literal representing the hour value to apply to the function |

| | | | |
|---|---|---|---|
| minute_integer_col | Y | integer | Name of column or Integer literal representing the minute value to apply to the function |
| second_integer_col | Y | integer | Name of column or Integer literal representing the second value to apply to the function |
| time_format_string | Y | string | String literal identifying the time format to apply to the value |

For more information on syntax standards, see *Language Documentation Syntax Notes*.

### hour_integer_col

Integer literal or name of the column containing integer values for the hour. Values must integers between 0 and 23, inclusive.

- Missing values for this function in the source data result in missing values in the output.
- Multiple columns and wildcards are not supported.

**Usage Notes:**

| Required? | Data Type | Example Value |
|---|---|---|
| Yes | Integer (literal or column reference) | 15 |

### minute_integer_col

Integer literal or name of the column containing integer values for the minutes. Values must integers between 0 and 59, inclusive.

- Missing values for this function in the source data result in missing values in the output.
- Multiple columns and wildcards are not supported.

**Usage Notes:**

| Required? | Data Type | Example Value |
|---|---|---|
| Yes | Integer (literal or column reference) | 23 |

### second_integer_col

Integer literal or name of the column containing integer values for the second. Values must integers between 0 and 59, inclusive.

- Missing values for this function in the source data result in missing values in the output.
- Multiple columns and wildcards are not supported.

**Usage Notes:**

| Required? | Data Type | Example Value |
|---|---|---|
| Yes | Integer (literal or column reference) | 45 |

### time_format_string

For more information, see *DATEFORMAT Function*.

# Examples

> **Tip:** For additional examples, see *Common Tasks*.

### Example - date and time functions

This example provides an overview on various date and time functions.

**Functions**:

| Item | Description |
|---|---|
| DATE Function | Generates a date value from three inputs of Integer type: year, month, and day. |
| TIME Function | Generates time values from three inputs of Integer type: hour, minute, and second. |
| DATETIME Function | Generates a Datetime value from the following inputs of Integer type: year, month, day, hour, minute, and second. |
| DATEFORMAT Function | Formats a specified Datetime set of values according to the specified date format. Source values can be a reference to a column containing Datetime values. |

**Source:**

| year | month | day | hour | minute | second |
|---|---|---|---|---|---|
| 2016 | 10 | 11 | 2 | 3 | 0 |
| 2015 | 11 | 20 | 15 | 22 | 30 |
| 2014 | 12 | 25 | 18 | 30 | 45 |

**Transformation:**

| Transformation Name | `New formula` |
|---|---|
| **Parameter: Formula type** | `Single row formula` |
| **Parameter: Formula** | `DATEFORMAT(DATE (year, month, day),'yyyy-MM-dd')` |
| **Parameter: New column name** | `'fctn_date'` |

| Transformation Name | `New formula` |
|---|---|
| **Parameter: Formula type** | `Single row formula` |
| **Parameter: Formula** | `DATEFORMAT(TIME (hour, minute, second),'HH-mm-ss')` |
| **Parameter: New column name** | `'fctn_time'` |

| Transformation Name | `New formula` |
|---|---|
| **Parameter: Formula type** | `Single row formula` |
| **Parameter: Formula** | `DATEFORMAT(DATETIME (year, month, day, hour, minute,` `second),'yyyy-MM-dd-HH:mm:ss')` |

| Parameter: New column name | `'fctn_datetime'` |
|---|---|

**Results:**

> **NOTE:** All inputs must be inferred as Integer type and must be valid values for the specified input. For example, month values must be integers between 1 and 12, inclusive.

| year | month | day | hour | minute | second | fctn_date | fctn_time | fctn_datetime |
|---|---|---|---|---|---|---|---|---|
| 2016 | 10 | 11 | 2 | 3 | 0 | 2016-10-11 | 02-03-00 | 2016-10-11-02:03:00 |
| 2015 | 11 | 20 | 15 | 22 | 30 | 2015-11-20 | 15-22-30 | 2015-11-20-15:22:30 |
| 2014 | 12 | 25 | 18 | 30 | 45 | 2014-12-25 | 18-30-45 | 2014-12-25-18:30:45 |

You can apply other date and time functions to the generated columns. For an example, see *YEAR Function*.

# DATETIME Function

**Contents:**

---

Generates a Datetime value from the following inputs of Integer type: year, month, day, hour, minute, and second.

- Source values can be Integer literals or column references to values that can be inferred as Integers.
- If any of the source values are invalid or out of range, a missing value is returned.
- This function must be nested within another function that accepts date values, such as DATEFORMAT, as arguments. See the example below.

**Wrangle vs. SQL:** This function is part of Wrangle , a proprietary data transformation language. Wrangle is not SQL. For more information, see *Wrangle Language*.

## Basic Usage

**Integer literal values:**

```
dateformat(datetime(2015,02,15, 2, 4, 0),&apos;yyyy-MM-dd:HH:mm:ss&apos;)
```

**Output:** Returns the formatted date value: `2015-02-15:02:04:00`.

**Column reference values:**

```
dateformat(datetime(myYear, myMonth, myDay, myHour, myMin, mySec),&apos;MMM yyyy - HH:MM:
SS&apos;)
```

**Output:** Generates date values where:

- `MMM` = short value for `myMonth`
- `yyyy` = value from `myYear`
- `HH` = value from `myHour`
- `MM` = value from `myMin`
- `SS` = value from `mySec`

## Syntax and Arguments

```
dateformat(datetime(year_integer_col,month_integer_col,day_Integer_col, hour_Integer_col,
minute_Integer_col,second_Integer_col  ),&apos;date_format_string&apos;)
```

| Argument | Required? | Data Type | Description |
|---|---|---|---|
| year_integer_col | Y | integer | Name of column or Integer literal representing the year value to apply to the function |
| month_integer_col | Y | integer | Name of column or Integer literal representing the month value to apply to the function |
| day_integer_col | Y | integer | Name of column or Integer literal representing the day value to apply to the function |
| hour_integer_col | Y | integer | Name of column or Integer literal representing the hour value to apply to the function |
| minute_integer_col | Y | integer | Name of column or Integer literal representing the minute value to apply to the function |
| second_integer_col | Y | integer | Name of column or Integer literal representing the day second to apply to the function |
| date_format_string | Y | string | String literal identifying the date format to apply to the value |

For more information on syntax standards, see *Language Documentation Syntax Notes*.

### year_integer_col

Integer literal or name of the column containing integer values for the year.

- Missing values for this function in the source data result in missing values in the output.
- Multiple columns and wildcards are not supported.

**Usage Notes:**

| Required? | Data Type | Example Value |
|---|---|---|
| Yes | Integer (literal or column reference) | 2015 |

### month_integer_col

Integer literal or name of the column containing integer values for the month.

- Values must be 1 or more, with a maximum value of 12.
- Missing values for this function in the source data result in missing values in the output.
- Multiple columns and wildcards are not supported.

**Usage Notes:**

| Required? | Data Type | Example Value |
|---|---|---|
| Yes | Integer (literal or column reference) | 4 |

### day_integer_col

Integer literal or name of the column containing integer values for the day.

- Values must be 1 or more, with a maximum value for any month of 31.
- Missing values for this function in the source data result in missing values in the output.
- Multiple columns and wildcards are not supported.

**Usage Notes:**

| Required? | Data Type | Example Value |
|-----------|-----------|---------------|
| Yes | Integer (literal or column reference) | 26 |

### hour_integer_col

Integer literal or name of the column containing integer values for the hour.

- Values must be 0 or more, with a maximum value for any hour of 23.
- Missing values for this function in the source data result in missing values in the output.
- Multiple columns and wildcards are not supported.

**Usage Notes:**

| Required? | Data Type | Example Value |
|-----------|-----------|---------------|
| Yes | Integer (literal or column reference) | 21 |

### minute_integer_col

Integer literal or name of the column containing integer values for the minute.

- Values must be 0 or more, with a maximum value for any minute of 59.
- Missing values for this function in the source data result in missing values in the output.
- Multiple columns and wildcards are not supported.

**Usage Notes:**

| Required? | Data Type | Example Value |
|-----------|-----------|---------------|
| Yes | Integer (literal or column reference) | 04 |

### second_integer_col

Integer literal or name of the column containing integer values for the second.

- Values must be 0 or more, with a maximum value for any second of 59.
- Missing values for this function in the source data result in missing values in the output.
- Multiple columns and wildcards are not supported.

**Usage Notes:**

| Required? | Data Type | Example Value |
|-----------|-----------|---------------|
| Yes | Integer (literal or column reference) | 15 |

### date_format_string

String value indicating the date format to apply to the input values.

Designer Cloud Powered by Trifacta Enterprise Edition supports Java formatting strings, with some exceptions.

> **NOTE:** If the platform cannot recognize the date format string, the generated result is written as a string value.

- Missing values for this function in the source data result in missing values in the output.

- Multiple columns and wildcards are not supported.

**Usage Notes:**

| Required? | Data Type | Example Value |
|-----------|-----------|---------------|
| Yes | String | `'MM/dd /yyyy'` |

## Examples

> **Tip:** For additional examples, see *Common Tasks*.

### Example - date and time functions

This example provides an overview on various date and time functions.

**Functions**:

| Item | Description |
|------|-------------|
| DATE Function | Generates a date value from three inputs of Integer type: year, month, and day. |
| TIME Function | Generates time values from three inputs of Integer type: hour, minute, and second. |
| DATETIME Function | Generates a Datetime value from the following inputs of Integer type: year, month, day, hour, minute, and second. |
| DATEFORMAT Function | Formats a specified Datetime set of values according to the specified date format. Source values can be a reference to a column containing Datetime values. |

**Source:**

| year | month | day | hour | minute | second |
|------|-------|-----|------|--------|--------|
| 2016 | 10 | 11 | 2 | 3 | 0 |
| 2015 | 11 | 20 | 15 | 22 | 30 |
| 2014 | 12 | 25 | 18 | 30 | 45 |

**Transformation:**

| Transformation Name | `New formula` |
|---------------------|---------------|
| **Parameter: Formula type** | `Single row formula` |
| **Parameter: Formula** | `DATEFORMAT(DATE (year, month, day),'yyyy-MM-dd')` |
| **Parameter: New column name** | `'fctn_date'` |

| Transformation Name | `New formula` |
|---------------------|---------------|
| **Parameter: Formula type** | `Single row formula` |
| **Parameter: Formula** | `DATEFORMAT(TIME (hour, minute, second),'HH-mm-ss')` |
| **Parameter: New column name** | `'fctn_time'` |

| | |
|---|---|
| **Transformation Name** | `New formula` |
| **Parameter: Formula type** | `Single row formula` |
| **Parameter: Formula** | `DATEFORMAT(DATETIME (year, month, day, hour, minute, second),'yyyy-MM-dd-HH:mm:ss')` |
| **Parameter: New column name** | `'fctn_datetime'` |

**Results:**

> **NOTE:** All inputs must be inferred as Integer type and must be valid values for the specified input. For example, month values must be integers between 1 and 12, inclusive.

| year | month | day | hour | minute | second | fctn_date | fctn_time | fctn_datetime |
|------|-------|-----|------|--------|--------|-----------|-----------|---------------|
| 2016 | 10 | 11 | 2 | 3 | 0 | 2016-10-11 | 02-03-00 | 2016-10-11-02:03:00 |
| 2015 | 11 | 20 | 15 | 22 | 30 | 2015-11-20 | 15-22-30 | 2015-11-20-15:22:30 |
| 2014 | 12 | 25 | 18 | 30 | 45 | 2014-12-25 | 18-30-45 | 2014-12-25-18:30:45 |

You can apply other date and time functions to the generated columns. For an example, see *YEAR Function*.

# DATEADD Function

**Contents:**

---

Add a specified number of units to a valid date. Units can be any supported Datetime unit (e.g. `minute`, `month`, `year`, etc.). Input must be a column reference containing dates.

> **NOTE:** If this function computes values out of the supported range of dates, the values are written as mismatched values, and the column is likely to be typed as a Datetime column.

**Wrangle vs. SQL:** This function is part of Wrangle , a proprietary data transformation language. Wrangle is not SQL. For more information, see *Wrangle Language*.

## Basic Usage

```
dateadd(myDate, 3, month)
```

**Output:** Returns the values in the `myDate` column with three months added to them.

> **NOTE:** Output dates are always formatted with dashes. For example, if the input values include `12/31/2016`, a `dateadd` function output might be `03-31-2017`.

## Syntax and Arguments

```
dateadd(date,delta,date_units)
```

| Argument | Required? | Data Type | Description |
|---|---|---|---|
| date | Y | datetime | Starting date to compare |
| delta | Y | integer | Number of units to apply to the date value. |
| date_units | Y | string | String literal representing the date units to use in the comparison |

For more information on syntax standards, see *Language Documentation Syntax Notes*.

**date**

Date values to which to add. It must be a column reference.

**Usage Notes:**

| Required? | Data Type | Example Value |
|---|---|---|
| Yes | String (Date column reference) | `LastContactDate` |

**delta**

Number of units to apply to the date values.

- Negative integer values are accepted.

**Usage Notes:**

| Required? | Data Type | Example Value |
|---|---|---|
| Yes | Integer | `-3` |

**date_units**

Unit of date measurement to which to apply the delta value.

**Usage Notes:**

| Required? | Data Type | Example Value |
|---|---|---|
| Yes | String | `year` |

**Accepted Value for date units:**

- `year`
- `month`
- `week`
- `day`
- `hour`
- `minute`
- `second`
- `millisecond`

## Examples

> **Tip:** For additional examples, see *Common Tasks*.

**Example - DATEADD Function**

**Source:**

Here are some example invoices and their dates. You want to calculate the 90-day and 180-day limits, at which point interest is charged.

| InvNum | InvDate |
|--------|---------|
| inv0001 | 1/1/2016 |
| inv0002 | 7/15/2016 |
| inv0003 | 12/30/2016 |

**Transformation:**

| Transformation Name | New formula |
|---|---|
| **Parameter: Formula type** | Single row formula |
| **Parameter: Formula** | dateadd(InvDate,90,day) |
| **Parameter: New column name** | 'plus90' |

| Transformation Name | New formula |
|---|---|
| **Parameter: Formula type** | Single row formula |
| **Parameter: Formula** | dateadd(InvDate,180,day) |
| **Parameter: New column name** | 'plus180' |

**Results:**

> **NOTE:** The output format is always formatted with dashes.

| InvNum | InvDate | plus90 | plus180 |
|--------|---------|--------|---------|
| inv0001 | 1/1/2016 | 3-31-2016 | 6-29-2016 |
| inv0002 | 7/15/2016 | 10-13-2016 | 1-11-2017 |
| inv0003 | 12/30/2016 | 3-30-2017 | 6-28-2017 |

# DATEDIF Function

**Contents:**

---

Calculates the difference between two valid date values for the specified units of measure.

- Inputs must be column references.
- The first value is used as the baseline to compare the date values.
- Results are calculated to the integer value that is closest to and lower than the exact total; remaining decimal values are dropped.

**Wrangle vs. SQL:** This function is part of Wrangle , a proprietary data transformation language. Wrangle is not SQL. For more information, see *Wrangle Language*.

## Basic Usage

```
datedif(StartDate, EndDate, month)
```

**Output:** Returns the number of full months that have elapsed between `StartDate` and `EndDate`.

## Syntax and Arguments

```
datedif(date1,date2,date_units)
```

| Argument | Required? | Data Type | Description |
|---|---|---|---|
| date1 | Y | datetime | Starting date to compare |
| date2 | Y | datetime | Ending date to compare |
| date_units | Y | string | String literal representing the date units to use in the comparison |

For more information on syntax standards, see *Language Documentation Syntax Notes*.

**date1, date2**

Date values to compare using the `date_units` units. If `date2` > `date1`, then results are positive.

- Date values must be column references.
- If `date1` and `date2` have a specified time zone offset, the function calculates the difference including the timezone offsets.
- If `date1` does not have a specified time zone but `date2` does, the function uses the local time in the same time zone as `date2` to calculate the difference. The functions returns the difference without the time zone offset.

**Usage Notes:**

| Required? | Data Type | Example Value |
|---|---|---|
| Yes | String (Date column reference) | `LastContactDate` |

**date_units**

Unit of date measurement to calculate between the two valid dates.

**Usage Notes:**

| Required? | Data Type | Example Value |
|---|---|---|
| Yes | String | `year` |

**Accepted Value for date units:**

- `year`
- `quarter`
- `month`
- `dayofyear`
- `week`
- `day`
- `hour`
- `minute`
- `second`
- `millisecond`

## Examples

> **Tip:** For additional examples, see *Common Tasks*.

**Example - aged orders**

This example illustrates how to calculate the number of days that have elapsed between the order date and today.

**Function**:

| Item | Description |
|---|---|
| DATEDIF Function | Calculates the difference between two valid date values for the specified units of measure. |
| TODAY Function | Derives the value for the current date in UTC time zone. You can specify a different time zone by optional parameter. |
| IF Function | The `IF` function allows you to build if/then/else conditional logic within your transforms. |

**Source:**

For the orders in the following set, you want to charge interest for those ones that are older than 90 days.

| OrderId | OrderDate | Amount |
|---|---|---|

| | | |
|------|----------|------|
| 1001 | 1/31/16  | 1000 |
| 1002 | 11/15/15 | 1000 |
| 1003 | 12/18/15 | 1000 |
| 1004 | 1/15/16  | 1000 |

**Transformation:**

The first step is to create a column containing today's (03/03/16) date value:

| | |
|---|---|
| **Transformation Name** | New formula |
| **Parameter: Formula type** | Single row formula |
| **Parameter: Formula** | TODAY() |
| **Parameter: New column name** | 'Today' |

You can now use this value as the basis for computing the number of elapsed days for each invoice:

| | |
|---|---|
| **Transformation Name** | New formula |
| **Parameter: Formula type** | Single row formula |
| **Parameter: Formula** | DATEDIF(OrderDate, Today, day) |

The age of each invoice in days is displayed in the new column. Now, you want to add a little bit of information to this comparison. Instead of just calculating the number of days, you could write out the action to undertake. Replace the above with the following:

| | |
|---|---|
| **Transformation Name** | New formula |
| **Parameter: Formula type** | Single row formula |
| **Parameter: Formula** | IF((DATEDIF(OrderDate, Today, day) > 90),'Charge interest','no action') |
| **Parameter: New column name** | 'TakeAction' |

To be fair to your customers, you might want to issue a notice at 45 days that the invoice is outstanding. You can replace the above with the following:

| | |
|---|---|
| **Transformation Name** | New formula |
| **Parameter: Formula type** | Single row formula |
| **Parameter: Formula** | IF(DATEDIF(OrderDate, Today, day) > 90,'Charge interest',IF (DATEDIF(OrderDate, Today, day) > 45),'Send letter','no action')) |
| **Parameter: New column name** | 'TakeAction' |

By using nested instances of the `IF` function, you can generate multiple results in the `TakeAction` column.

For the items that are over 90 days old, you want to charge 5% interest. You can do the following:

| Transformation Name | Edit column with formula |
|---|---|
| Parameter: Columns | Amount |
| Parameter: Formula | IF(TakeAction == 'Charge interest',Amount * 1.05, Amount) |

The above sets the value in the `Amount` column based on the conditional of whether the `TakeAction` column value is `Charge interest`. If so, apply 5% interest to the value in the `Amount` column.

**Results:**

| OrderId | OrderDate | Amount | Today | TakeAction |
|---|---|---|---|---|
| 1001 | 1/31/16 | 1000 | 03/03/16 | no action |
| 1002 | 11/15/15 | 1050 | 03/03/16 | Charge interest |
| 1003 | 12/18/15 | 1000 | 03/03/16 | Send letter |
| 1004 | 1/15/16 | 1000 | 03/03/16 | Send letter |

### Example - dayofyear Calculations

This example demonstrates how `dayofyear` is calculated using the `DATEDIF` function, specifically how leap years and leap days are handled. Below, you can see some example dates. The year 2012 was a leap year.

**Source:**

| dateId | d1 | d2 | Notes |
|---|---|---|---|
| 1 | 1/1/10 | 10/10/10 | Same year; no leap year |
| 2 | 1/1/10 | 10/10/11 | Different years; no leap year |
| 3 | 10/10/11 | 1/1/10 | Reverse dates of previous row |
| 4 | 2/28/11 | 4/1/11 | Same year; no leap year; |
| 5 | 2/28/12 | 4/1/12 | Same year; leap year; spans leap day |
| 6 | 2/29/12 | 4/1/12 | Same year; leap year; d1 = leap day |
| 7 | 2/28/11 | 2/29/12 | Diff years; d2 = leap day; converted to March 1 in d1 year |

**Transformation:**

In this case, the transform is simple:

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | datedif(d1,d2,dayofyear) |
| Parameter: New column name | 'datedifs' |

**Results:**

| dateId | d1 | d2 | datedifs | Notes |
|---|---|---|---|---|

| 1 | 1/1/10 | 10/10/10 | 282 | Same year; no leap year |
|---|--------|----------|-----|--------------------------|
| 2 | 1/1/10 | 10/10/11 | 282 | Different years; no leap year |
| 3 | 10/10/11 | 1/1/10 | -282 | Reverse dates of previous row |
| 4 | 2/28/11 | 4/1/11 | 32 | Same year; no leap year; |
| 5 | 2/28/12 | 4/1/12 | 33 | Same year; leap year; spans leap day |
| 6 | 2/29/12 | 4/1/12 | 32 | Same year; leap year; d1 = leap day |
| 7 | 2/28/11 | 2/29/12 | 1 | Diff years; d2 = leap day; converted to March 1 in d1 year |

**Rows 1 - 3:**

- Row 1 provides the baseline calc.
- In Row 2, the same days of the year are used, but the year is different by a count of 1. However, since we are computing `dayofyear` the result is the same as for Row 1.

> **NOTE:** When computing `dayofyear`, the year value for d2 is converted to the year of d1. The difference is then computed.

- Row 3 represents the reversal of dates in Row 2.

> **NOTE:** Negative values for a `dayofyear` calculation indicate that d2 occurs earlier in the calendar than d1, ignoring year.

**Rows 4 - 7: Leap years**

- Row 4 provides a baseline calculation for a non-leap year.
- Row 5 uses the same days of year as Row 4, but the year (2012) is a leap year. Dates span a leap date (February 29). Note that the `DATEDIF` result is 1 more than the value in the previous row.

> **NOTE:** When the two dates span a leap date and the year for d1 is a leap year, then February 29 is included as part of the calculated result.

- Row 6 moves date 1 forward by one day, so it is now on a leap day date. Result is one less than the previous row, which also spanned leap date.
- Row 7 switches the leap date to d2. In this case, d2 is converted to the year of d1. However, since it was a leap day originally, in the year of d1, this value is March 1. Thus, the difference between the two dates is 1.

> **NOTE:** If d2 is a leap date and the year for d1 is not a leap year, the date used in for d2 is March 1 in the year of d1.

# DATEFORMAT Function

**Contents:**

---

Formats a specified Datetime set of values according to the specified date format. Source values can be a reference to a column containing Datetime values.

- If the source Datetime value does not include a valid input for this function, a missing value is returned.
- Designer Cloud Powered by Trifacta® Enterprise Edition supports a wide variety of formats for Datetime fields.
- You can explore the available Datetime formats through the Transformer page. From a column's type drop-down, select **Date/Time** . Then, select the formatting category. From the displayed drop-down, you can select a specific format. When this transform step is added to your recipe, you can edit it to see how the format is specified in Wrangle .

For more information on formatting numeric types, see *NUMFORMAT Function*.

**Wrangle vs. SQL:** This function is part of Wrangle , a proprietary data transformation language. Wrangle is not SQL. For more information, see *Wrangle Language*.

## Basic Usage

```
dateformat(MyDate, &apos;yyyy-MM-dd&apos;)
```

**Output:** Returns the valid date values in the `MyDate` column converted to year-month-day format.

## Syntax and Arguments

```
dateformat(Datetime_col, date_format_string)
```

| Argument | Required? | Data Type | Description |
|----------|-----------|-----------|-------------|
| Datetime_col | Y | datetime | Name of column containing date values to be formatted |
| date_format_string | Y | string | String literal identifying the date format to apply to the value |

For more information on syntax standards, see *Language Documentation Syntax Notes*.

**datetime_col**

Name of the column whose date data is to be formatted.

- Missing values for this function in the source data result in missing values in the output.

- Multiple columns and wildcards are not supported.

**Usage Notes:**

| Required? | Data Type | Example Value |
|-----------|-----------|---------------|
| Yes | Datetime | `myDate` |

### date_format_string

String value indicating the date format to apply to the input values.

Designer Cloud Powered by Trifacta Enterprise Edition supports Java formatting strings, with some exceptions.

> **NOTE:** Two-digit values for the year that are older than 80 years from the current year are forward-ported into the future. For example, in a job run on Dec 31, 2021, the date `01/01/41` is interpreted as 01/01 /1941. However, if the job is run the next day (January 01, 2022), then the same data is interpreted as 01 /01/2041. For more information including workarounds, see *Datetime Data Type*.

> **NOTE:** If the platform cannot recognize the date format string, the generated result is written as a string value.

- Missing values for this function in the source data result in missing values in the output.
- Multiple columns and wildcards are not supported.

**Usage Notes:**

| Required? | Data Type | Example Value |
|-----------|-----------|---------------|
| Yes | String | `'MM/dd/yyyy'` |

## Examples

> **Tip:** For additional examples, see *Common Tasks*.

### Example - formatting date values

This example illustrates several ways of wrangling heterogeneous date values, including the use of the `DATEFORM AT` function.

**Source:**

Your dataset includes the following messy date values:

| MyDate |
|--------|
| 2/1/00 9:20 |
| 4/5/10 11:25 |

| | |
|---|---|
| 6/7/99 22:00 | |
| 12/20/1894 15:45:00 | |
| 13/7/1999 22:00:00 | |

**Transformation:**

When this data is loaded into the application, it is not immediately recognized as a Datetime column, as the variation among the data complicates deciding on the proper date format. The first three rows look to be in a consistent format, but the other two are problematic.

You can try to change the column to a Datetime type with a format that matches the first three rows. You can select the appropriate format through the type drop-down. When previewed, the transform looks like the following:

> **NOTE:** Do not add this transform at this time. It is strictly used for reviewing the effects on data quality.

| | |
|---|---|
| **Transformation Name** | `Change column data type` |
| **Parameter: Columns** | `MyDate` |
| **Parameter: New type** | `Custom or Date/Time` |
| **Parameter: Specify type** | `'mm-dd-yy hh:mm:ss','mm*dd*yy*HH:MM'` |

When the column is reformatted, you should notice that the last two values in the column are mismatched. In the column histogram, you can see that date ranges include the 1999 date in the third row, so the final row should work if it was a valid date.

The 1894 value looks like an outlier value and could be removed:

| | |
|---|---|
| **Transformation Name** | `Filter rows` |
| **Parameter: Condition** | `Custom formula` |
| **Parameter: Type of formula** | `Custom single` |
| **Parameter: Condition** | `matches([MyDate], `12/20/1894`)` |
| **Parameter: Action** | `Delete matching rows` |

For the remaining 1999 row, you can delete it or use the following transforms to conform it to the other rows. Use the following transform to change the `13` month value to a `12`:

| | |
|---|---|
| **Transformation Name** | `Replace text or pattern` |
| **Parameter: Column** | `MyDate` |
| **Parameter: Find** | `` `13/` `` |
| **Parameter: Replace with** | `'12\/'` |
| **Parameter: Match all occurrences** | `true` |

The following two transforms complete the cleanup steps:

| | |
|---|---|
| **Transformation Name** | `Replace text or pattern` |

| Parameter: Column | MyDate |
|---|---|
| Parameter: Find | `` `/1999` `` |
| Parameter: Replace with | `'\/99'` |
| Parameter: Match all occurrences | `true` |

| Transformation Name | Replace text or pattern |
|---|---|
| Parameter: Column | MyDate |
| Parameter: Find | `` `:#+:00` `` |
| Parameter: Replace with | `':00'` |
| Parameter: Match all occurrences | `true` |

If you apply the original formatting step, all dates are valid:

| Transformation Name | Change column data type |
|---|---|
| Parameter: Columns | MyDate |
| Parameter: New type | Custom or Date/Time |
| Parameter: Specify type | `'mm-dd-yy hh:mm:ss','mm*dd*yy*HH:MM'` |

Now, your Datetime column can be formatted as needed using the `dateformat` function. The following step generates a new column that contains year, month, and day information as a single numeric value:

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | `dateformat(MyDate, 'yyyyMMdd')` |

**Results:**

The final dataset should look like the following:

| MyDate | dateformat_MyDate |
|---|---|
| 2/1/00 9:20 | 20000201 |
| 4/5/10 11:25 | 20100405 |
| 6/7/99 22:00 | 19990607 |
| 12/7/99 22:00 | 19991207 |

### Example - Other date formatting variations

**Numeric date, year first**

| Source | Transformation | Results |
|---|---|---|
| | | |

| 2/15/16 13:26:58.123 | | | 2016-02-15 |
|---|---|---|---|
| 3/12/99 2:45:21.456 | **Transformation Name** | New formula | 1999-03-12 |
| 11/21/11 23:02:18.000 | **Parameter: Formula type** | Single row formula | 2011-11-21 |
| | **Parameter: Formula** | dateformat(Timestamp,'yyyy-MM-dd') | |
| | **Parameter: New column name** | 'newTimestamp' | |

**Numeric date, American style**

| Source | Transformation | | Results |
|---|---|---|---|
| 2/15/16 13:26:58.123 | | | 2/15/16 |
| 3/12/99 2:45:21.456 | **Transformation Name** | New formula | 3/12/99 |
| 11/21/11 23:02:18.000 | **Parameter: Formula type** | Single row formula | 11/21/11 |
| | **Parameter: Formula** | dateformat(Timestamp,'M/d/yy') | |
| | **Parameter: New column name** | 'newTimestamp' | |

**Full written date**

| Source | Transformation | | Results |
|---|---|---|---|
| 2/15/16 13:26:58.123 | | | February 15, 2016 |
| 3/12/99 2:45:21.456 | **Transformation Name** | New formula | March 12, 1999 |
| 11/21/11 23:02:18.000 | **Parameter: Formula type** | Single row formula | November 21, 2011 |
| | **Parameter: Formula** | dateformat(Timestamp,'MMMM dd, yyyy') | |
| | **Parameter: New column name** | 'newTimestamp' | |

**Abbreviated date, including abbreviated day of week**

| Source | Transformation | | Results |
|---|---|---|---|
| 2/15/16 13:26:58.123 | | | Mon Feb 15, 2016 |
| 3/12/99 2:45:21.456 | **Transformation Name** | New formula | Fri Mar 12, 1999 |
| 11/21/11 23:02:18.000 | **Parameter: Formula type** | Single row formula | Mon Nov 21, 2011 |
| | **Parameter: Formula** | dateformat(Timestamp,'EEE MMM dd, yyyy') | |
| | **Parameter: New column name** | 'newTimestamp' | |

**Full 24-hour time**

| Source | Transformation | | Results |
|--------|----------------|--|---------|
| 2/15/16 13:26: 58.123<br><br>3/12/99 2:45: 21.456<br><br>11/21/11 23:02: 18.000 | **Transformation Name** | `New formula` | 13:26: 58.123<br><br>2:45: 21.456<br><br>23:02: 18.000 |
| | **Parameter: Formula type** | `Single row formula` | |
| | **Parameter: Formula** | `dateformat(Timestamp,'HH:mm:ss.SSS')` | |
| | **Parameter: New column name** | `'newTimestamp'` | |

**Twelve-hour time with AM/PM indicator**

| Source | Transformation | | Results |
|--------|----------------|--|---------|
| 2/15/16 13:26: 58.123<br><br>3/12/99 2:45: 21.456<br><br>11/21/11 23:02: 18.000 | **Transformation Name** | `New formula` | 1:26:58 PM<br><br>2:45:21 AM<br><br>11:02:18 PM |
| | **Parameter: Formula type** | `Single row formula` | |
| | **Parameter: Formula** | `dateformat(Timestamp,'h:mm:ss a')` | |
| | **Parameter: New column name** | `'newTimestamp'` | |
| | **NOTE:** For this function, use of the lower-case hour indicator (h or hh) requires the use of an AM/PM indicator (a). | | |

# UNIXTIMEFORMAT Function

**Contents:**

- *Basic Usage*
- *Syntax and Arguments*
    - *unixtime_col*
    - *date_format_string*
- *Examples*
    - *Example - Unix timestamp formatting variations*

Formats a set of Unix timestamps according to a specified date formatting string.

Source values can be a reference to a column containing Unix timestamp values.

> **NOTE:** Date values must be converted to Unix timestamps before applying this function. Unix time measures the number of milliseconds that have elapsed since January 1, 1970 00:00:00 (UTC). See *UNIXTIME Function.*

Supported format strings for this function are the same as the supported format strings for the `DATEFORMAT` function.

- For more information on formatting Unix or standard date formats, see *DATEFORMAT Function*.
- For more information on formatting numeric types, see *NUMFORMAT Function*.

**Wrangle vs. SQL:** This function is part of Wrangle , a proprietary data transformation language. Wrangle is not SQL. For more information, see *Wrangle Language*.

## Basic Usage

```
unixtimeformat(MyUnixDate, &apos;yyyy-MM-dd&apos;)
```

**Output:** Returns the Unix timestamp values in the `MyUnixDate` column, converted to year-month-day format.

## Syntax and Arguments

```
unixtimeformat(unixtime_col, date_format_string)
```

| Argument | Required? | Data Type | Description |
|---|---|---|---|
| unixtime_col | Y | datetime | Name of column whose Unix timestamp values are to be formatted |
| date_format_string | Y | string | String literal identifying the date format to apply to the value |

For more information on syntax standards, see *Language Documentation Syntax Notes*.

**unixtime_col**

Name of the column whose Unix time data is to be formatted.

- Missing values for this function in the source data result in missing values in the output.

- Multiple columns and wildcards are not supported.

**Usage Notes:**

| Required? | Data Type | Example Value |
|-----------|-----------|---------------|
| Yes | Datetime (formatted as Unix time integer values) | `myDate` |

### date_format_string

String value indicating the date format to apply to the input values.

> **NOTE:** If the platform cannot recognize the date format string, the generated result is written as a string value.

For more information on the supported formatting strings, see below.

- Missing values for this function in the source data result in missing values in the output.
- Multiple columns and wildcards are not supported.

**Usage Notes:**

| Required? | Data Type | Example Value |
|-----------|-----------|---------------|
| Yes | String | `'MM/dd/yyyy'` |

## Examples

> **Tip:** For additional examples, see *Common Tasks*.

**Example - Unix timestamp formatting variations**

| Description | unixTimestamp column | Transform | | newUnixTimestamp column |
|-------------|----------------------|-----------|---|-------------------------|
| Numeric date, year first | 1454946120000<br><br>1451433600000<br><br>1430032020000 | **Transformation Name** | `New formula` | 2016-02-08<br><br>2015-12-30<br><br>2015-04-26 |
| | | **Parameter: Formula type** | `Single row formula` | |
| | | **Parameter: Formula** | `unixtimeformat (unixTimestamp,'yyyy-MM-dd')` | |
| | | **Parameter: New column name** | `'newUnixTimestamp'` | |

| Numeric date, American style | 1454946120000  1451433600000  1430032020000 | | | 2/8/16  12/30/15  4/26/15 |
|---|---|---|---|---|
| | | **Transformation Name** | `New formula` | |
| | | **Parameter: Formula type** | `Single row formula` | |
| | | **Parameter: Formula** | `unixtimeformat (unixTimestamp,'M/d /yy')` | |
| | | **Parameter: New column name** | `'newUnixTimestamp'` | |
| Full written date | 1454946120000  1451433600000  1430032020000 | | | February 08, 2016  December 30, 2015  April 26, 2015 |
| | | **Transformation Name** | `New formula` | |
| | | **Parameter: Formula type** | `Single row formula` | |
| | | **Parameter: Formula** | `unixtimeformat (unixTimestamp,'MMMM dd, yyyy')` | |
| | | **Parameter: New column name** | `'newUnixTimestamp'` | |
| Abbreviated date, including abbreviated day of week | 1454946120000  1451433600000  1430032020000 | | | Mon Feb 08, 2016  Wed Dec 30, 2015  Sun Apr 26, 2015 |
| | | **Transformation Name** | `New formula` | |
| | | **Parameter: Formula type** | `Single row formula` | |
| | | **Parameter: Formula** | `unixtimeformat (unixTimestamp,'EEE MMM dd, yyyy')` | |
| | | **Parameter: New column name** | `'newUnixTimestamp'` | |
| Full 24-hour time | 1454946120000  1451433600000  1430032020000 | | | 15:42:00.000  00:00:00.000  07:07:00.00 |
| | | **Transformation Name** | `New formula` | |
| | | **Parameter: Formula type** | `Single row formula` | |
| | | **Parameter: Formula** | `unixtimeformat (unixTimestamp,'HH:mm: ss.SSS')` | |
| | | **Parameter: New column name** | `'newUnixTimestamp'` | |

| Twelve-hour time with AM/PM indicator | 1454946120000<br><br>1451433600000<br><br>1430032020000 | | | 3:42:00 PM<br><br>12:00:00 AM<br><br>7:07:00 AM |
|---|---|---|---|---|
| | | **Transformation Name** | `New formula` | |
| | | **Parameter: Formula type** | `Single row formula` | |
| | | **Parameter: Formula** | `unixtimeformat (unixTimestamp,'h:mm: ss a')` | |
| | | **Parameter: New column name** | `'newUnixTimestamp'` | |
| | | **NOTE:** For this function, use of the lower-case hour indicator (h or hh) requires the use of an AM/PM indicator (a). | | |

# MONTH Function

Derives the month integer value from a Datetime value. Source value can be a a reference to a column containing Datetime values or a literal.

> **NOTE:** If the source Datetime value does not include a valid input for this function, a missing value is returned.

**Wrangle vs. SQL:** This function is part of Wrangle , a proprietary data transformation language. Wrangle is not SQL. For more information, see *Wrangle Language*.

## Basic Usage

```
month(MyDate)
```

**Output:** Returns the numeric month values from the `MyDate` column.

## Syntax and Arguments

```
month(datetime_col)
```

| Argument | Required? | Data Type | Description |
|----------|-----------|-----------|-------------|
| datetime_col | Y | datetime | Name of column whose month values are to be computed |

For more information on syntax standards, see *Language Documentation Syntax Notes*.

**datetime_col**

Name of the column whose month value is to be computed.

- Missing values for this function in the source data result in missing values in the output.
- Multiple columns and wildcards are not supported.

**Usage Notes:**

| Required? | Data Type | Example Value |
|-----------|-----------|---------------|
| Yes | Datetime | `myDate` |

## Examples

> **Tip:** For additional examples, see *Common Tasks*.

**Example - Date element functions**

This example illustrates to how to use date-related functions to derive specific values for a Datetime column type.

**Functions**:

| Item | Description |
|---|---|
| YEAR Function | Derives the four-digit year value from a Datetime value. Source value can be a a reference to a column containing Datetime values or a literal. |
| MONTH Function | Derives the month integer value from a Datetime value. Source value can be a a reference to a column containing Datetime values or a literal. |
| MONTHNAME Function | Derives the full name from a Datetime value of the corresponding month as a String. Source value can be a reference to a column containing Datetime values or a literal. |
| WEEKDAYNAME Function | Derives the full name from a Datetime value of the corresponding weekday as a String. Source value can be a reference to a column containing Datetime values or a literal. |
| DAY Function | Derives the numeric day value from a Datetime value. Source value can be a a reference to a column containing Datetime values or a literal. |
| HOUR Function | Derives the hour value from a Datetime value. Generated hours are expressed according to the 24-hour clock. |
| MINUTE Function | Derives the minutes value from a Datetime value. Minutes are expressed as integers from 0 to 59. |
| SECOND Function | Derives the seconds value from a Datetime value. Source value can be a a reference to a column containing Datetime values or a literal. |

**Source:**

| date |
|---|
| 2/8/16 15:41 |
| 12/30/15 0:00 |
| 4/26/15 7:07 |

**Transformation:**

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | YEAR (date) |

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | MONTH (date) |

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | MONTHNAME (date) |

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |

| Parameter: Formula | WEEKDAYNAME (date) |
|---|---|

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | DAY (date) |

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | HOUR (date) |

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | MINUTE (date) |

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | SECOND (date) |

**Results:**

> **NOTE:** If the source Datetime value does not contain a valid input for one of these functions, no value is returned. See the `second_date` column below.

| date | year_date | month_date | monthname_date | weekdayname_date | day_date | hour_date | minute_date | secon |
|---|---|---|---|---|---|---|---|---|
| 2/8/16 15:41 | 2016 | 2 | February | Monday | 8 | 15 | 41 | |
| 12/30 /15 0: 00 | 2015 | 12 | December | Wednesday | 30 | 0 | 0 | |
| 4/26 /15 7: 07 | 2015 | 4 | April | Sunday | 26 | 7 | 7 | |

# MONTHNAME Function

Derives the full name from a Datetime value of the corresponding month as a String. Source value can be a reference to a column containing Datetime values or a literal.

> **NOTE:** If the source Datetime value does not include a valid input for this function, a missing value is returned.

**Wrangle vs. SQL:** This function is part of Wrangle , a proprietary data transformation language. Wrangle is not SQL. For more information, see *Wrangle Language*.

## Basic Usage

```
monthname(MyDate)
```

**Output:** Returns the month name from the `MyDate` column.

## Syntax and Arguments

```
<span>monthname</span><span>(datetime_col)</span>
```

| Argument | Required? | Data Type | Description |
|---|---|---|---|
| datetime_col | Y | datetime | Name of column whose month name values are to be computed |

For more information on syntax standards, see *Language Documentation Syntax Notes*.

**datetime_col**

Name of the column whose month name value is to be computed.

- Missing values for this function in the source data result in missing values in the output.
- Multiple columns and wildcards are not supported.

**Usage Notes:**

| Required? | Data Type | Example Value |
|---|---|---|
| Yes | Datetime | `myDate` |

## Examples

> **Tip:** For additional examples, see *Common Tasks*.

**Example - Date element functions**

This example illustrates to how to use date-related functions to derive specific values for a Datetime column type.

**Functions**:

| Item | Description |
|------|-------------|
| YEAR Function | Derives the four-digit year value from a Datetime value. Source value can be a a reference to a column containing Datetime values or a literal. |
| MONTH Function | Derives the month integer value from a Datetime value. Source value can be a a reference to a column containing Datetime values or a literal. |
| MONTHNAME Function | Derives the full name from a Datetime value of the corresponding month as a String. Source value can be a reference to a column containing Datetime values or a literal. |
| WEEKDAYNAME Function | Derives the full name from a Datetime value of the corresponding weekday as a String. Source value can be a reference to a column containing Datetime values or a literal. |
| DAY Function | Derives the numeric day value from a Datetime value. Source value can be a a reference to a column containing Datetime values or a literal. |
| HOUR Function | Derives the hour value from a Datetime value. Generated hours are expressed according to the 24-hour clock. |
| MINUTE Function | Derives the minutes value from a Datetime value. Minutes are expressed as integers from 0 to 59. |
| SECOND Function | Derives the seconds value from a Datetime value. Source value can be a a reference to a column containing Datetime values or a literal. |

**Source:**

| date |
|------|
| 2/8/16 15:41 |
| 12/30/15 0:00 |
| 4/26/15 7:07 |

**Transformation:**

| | |
|---|---|
| **Transformation Name** | New formula |
| **Parameter: Formula type** | Single row formula |
| **Parameter: Formula** | YEAR (date) |

| | |
|---|---|
| **Transformation Name** | New formula |
| **Parameter: Formula type** | Single row formula |
| **Parameter: Formula** | MONTH (date) |

| | |
|---|---|
| **Transformation Name** | New formula |
| **Parameter: Formula type** | Single row formula |
| **Parameter: Formula** | MONTHNAME (date) |

| | |
|---|---|
| **Transformation Name** | New formula |

| Parameter: Formula type | Single row formula |
|---|---|
| Parameter: Formula | WEEKDAYNAME (date) |

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | DAY (date) |

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | HOUR (date) |

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | MINUTE (date) |

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | SECOND (date) |

**Results:**

> **NOTE:** If the source Datetime value does not contain a valid input for one of these functions, no value is returned. See the `second_date` column below.

| date | year_date | month_date | monthname_date | weekdayname_date | day_date | hour_date | minute_date | secor |
|---|---|---|---|---|---|---|---|---|
| 2/8/16 15:41 | 2016 | 2 | February | Monday | 8 | 15 | 41 | |
| 12/30 /15 0: 00 | 2015 | 12 | December | Wednesday | 30 | 0 | 0 | |
| 4/26 /15 7: 07 | 2015 | 4 | April | Sunday | 26 | 7 | 7 | |

# EOMONTH Function

**Contents:**

- *Basic Usage*
- *Syntax and Arguments*
  - *start_date*
  - *delta_months*
- *Examples*
  - *Example - EOMONTH Function*

---

Returns the serial date number for the last day of the month before or after a specified number of months from a starting date.

**Wrangle vs. SQL:** This function is part of Wrangle , a proprietary data transformation language. Wrangle is not SQL. For more information, see *Wrangle Language*.

## Basic Usage

```
eomonth(myDate, 3)
```

**Output:** Returns the values for the last date of the month that is three months after the serial date number value in the `myDate` column.

## Syntax and Arguments

```
eomonth(start_date,delta_months)
```

| Argument | Required? | Data Type | Description |
|---|---|---|---|
| start_date | Y | Datetime | Starting date. |
| delta_months | Y | integer | Number of months before or after the starting date to apply to the date value. |

For more information on syntax standards, see *Language Documentation Syntax Notes*.

**start_date**

Starting date values from which to compute end-of-month values.

**Usage Notes:**

| Required? | Data Type | Example Value |
|---|---|---|
| Yes | Literal Datetime or column reference | `LastContactDate` |

**delta_months**

Number of months to add to the date value to determine the month whose last date is returned.

- Negative integer values are accepted.

**Usage Notes:**

| Required? | Data Type | Example Value |
|-----------|-----------|---------------|
| Yes | Integer | – 3 |

## Examples

> **Tip:** For additional examples, see *Common Tasks.*

### Example - EOMONTH Function

**Source:**

In the following table, you can see how the the end-of-month values are calculated from a baseline date of June 15, 2020.

| eventId | eventDate | deltaMonth |
|---------|-----------|------------|
| 1 | 06/15/2020 | -24 |
| 2 | 06/15/2020 | -18 |
| 3 | 06/15/2020 | -12 |
| 4 | 06/15/2020 | -6 |
| 5 | 06/15/2020 | -3 |
| 6 | 06/15/2020 | -2 |
| 7 | 06/15/2020 | -1 |
| 8 | 06/15/2020 | 0 |
| 9 | 06/15/2020 | 1 |
| 10 | 06/15/2020 | 2 |
| 11 | 06/15/2020 | 3 |
| 12 | 06/15/2020 | 6 |
| 13 | 06/15/2020 | 12 |
| 14 | 06/15/2020 | 18 |
| 15 | 06/15/2020 | 24 |

**Transformation:**

| Transformation Name | `New formula` |
|---------------------|-----------------|
| **Parameter: Formula type** | `Single row formula` |
| **Parameter: Formula** | `eomonth(eventDate,deltaMonth)` |

| Parameter: New column name | 'deltaMonthSerialDate' |
|---|---|

**Results:**

| eventId | eventDate | deltaMonth | deltaMonthSerialDate |
|---|---|---|---|
| 1 | 06/15/2020 | -24 | 43281 |
| 2 | 06/15/2020 | -18 | 43465 |
| 3 | 06/15/2020 | -12 | 43646 |
| 4 | 06/15/2020 | -6 | 43830 |
| 5 | 06/15/2020 | -3 | 43921 |
| 6 | 06/15/2020 | -2 | 43951 |
| 7 | 06/15/2020 | -1 | 43982 |
| 8 | 06/15/2020 | 0 | 44012 |
| 9 | 06/15/2020 | 1 | 44043 |
| 10 | 06/15/2020 | 2 | 44074 |
| 11 | 06/15/2020 | 3 | 44104 |
| 12 | 06/15/2020 | 6 | 44196 |
| 13 | 06/15/2020 | 12 | 44377 |
| 14 | 06/15/2020 | 18 | 44561 |
| 15 | 06/15/2020 | 24 | 44742 |

# YEAR Function

Derives the four-digit year value from a Datetime value. Source value can be a a reference to a column containing Datetime values or a literal.

> **NOTE:** If the source Datetime value does not include a valid input for this function, a missing value is returned.

**Wrangle vs. SQL:** This function is part of Wrangle , a proprietary data transformation language. Wrangle is not SQL. For more information, see *Wrangle Language*.

## Basic Usage

```
year(MyDate)
```

**Output:** Returns the four-digit year values from the `MyDate` column.

## Syntax and Arguments

```
year(datetime_col)
```

| Argument | Required? | Data Type | Description |
|---|---|---|---|
| datetime_col | Y | datetime | Name of column whose year values are to be computed |

For more information on syntax standards, see *Language Documentation Syntax Notes*.

**datetime_col**

Name of the column whose year value is to be computed.

- Missing values for this function in the source data result in missing values in the output.
- Multiple columns and wildcards are not supported.

**Usage Notes:**

| Required? | Data Type | Example Value |
|---|---|---|
| Yes | Datetime | `myDate` |

## Examples

> **Tip:** For additional examples, see *Common Tasks*.

**Example - Date element functions**

This example illustrates to how to use date-related functions to derive specific values for a Datetime column type.

**Functions**:

| Item | Description |
|---|---|
| YEAR Function | Derives the four-digit year value from a Datetime value. Source value can be a a reference to a column containing Datetime values or a literal. |
| MONTH Function | Derives the month integer value from a Datetime value. Source value can be a a reference to a column containing Datetime values or a literal. |
| MONTHNAME Function | Derives the full name from a Datetime value of the corresponding month as a String. Source value can be a reference to a column containing Datetime values or a literal. |
| WEEKDAYNAME Function | Derives the full name from a Datetime value of the corresponding weekday as a String. Source value can be a reference to a column containing Datetime values or a literal. |
| DAY Function | Derives the numeric day value from a Datetime value. Source value can be a a reference to a column containing Datetime values or a literal. |
| HOUR Function | Derives the hour value from a Datetime value. Generated hours are expressed according to the 24-hour clock. |
| MINUTE Function | Derives the minutes value from a Datetime value. Minutes are expressed as integers from 0 to 59. |
| SECOND Function | Derives the seconds value from a Datetime value. Source value can be a a reference to a column containing Datetime values or a literal. |

**Source:**

| date |
|---|
| 2/8/16 15:41 |
| 12/30/15 0:00 |
| 4/26/15 7:07 |

**Transformation:**

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | YEAR (date) |

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | MONTH (date) |

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | MONTHNAME (date) |

| Transformation Name | New formula |
|---|---|

| Parameter: Formula type | Single row formula |
|---|---|
| Parameter: Formula | WEEKDAYNAME (date) |

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | DAY (date) |

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | HOUR (date) |

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | MINUTE (date) |

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | SECOND (date) |

**Results:**

> **NOTE:** If the source Datetime value does not contain a valid input for one of these functions, no value is returned. See the `second_date` column below.

| date | year_date | month_date | monthname_date | weekdayname_date | day_date | hour_date | minute_date | secor |
|---|---|---|---|---|---|---|---|---|
| 2/8/16 15:41 | 2016 | 2 | February | Monday | 8 | 15 | 41 | |
| 12/30 /15 0: 00 | 2015 | 12 | December | Wednesday | 30 | 0 | 0 | |
| 4/26 /15 7: 07 | 2015 | 4 | April | Sunday | 26 | 7 | 7 | |

# DAY Function

Derives the numeric day value from a Datetime value. Source value can be a a reference to a column containing Datetime values or a literal.

> **NOTE:** If the source Datetime value does not include a valid input for this function, a missing value is returned.

**Wrangle vs. SQL:** This function is part of Wrangle , a proprietary data transformation language. Wrangle is not SQL. For more information, see *Wrangle Language*.

## Basic Usage

```
day(MyDate)
```

**Output:** Returns the day values from the `MyDate` column.

## Syntax and Arguments

```
day(datetime_col)
```

| Argument | Required? | Data Type | Description |
|---|---|---|---|
| datetime_col | Y | datetime | Name of column whose year values are to be computed |

For more information on syntax standards, see *Language Documentation Syntax Notes*.

**datetime_col**

Name of the column whose day value is to be computed.

- Missing values for this function in the source data result in missing values in the output.
- Multiple columns and wildcards are not supported.

**Usage Notes:**

| Required? | Data Type | Example Value |
|---|---|---|
| Yes | Datetime | `myDate` |

## Examples

> **Tip:** For additional examples, see *Common Tasks*.

**Example - Date element functions**

This example illustrates to how to use date-related functions to derive specific values for a Datetime column type.

**Functions**:

| Item | Description |
|---|---|
| YEAR Function | Derives the four-digit year value from a Datetime value. Source value can be a a reference to a column containing Datetime values or a literal. |
| MONTH Function | Derives the month integer value from a Datetime value. Source value can be a a reference to a column containing Datetime values or a literal. |
| MONTHNAME Function | Derives the full name from a Datetime value of the corresponding month as a String. Source value can be a reference to a column containing Datetime values or a literal. |
| WEEKDAYNAME Function | Derives the full name from a Datetime value of the corresponding weekday as a String. Source value can be a reference to a column containing Datetime values or a literal. |
| DAY Function | Derives the numeric day value from a Datetime value. Source value can be a a reference to a column containing Datetime values or a literal. |
| HOUR Function | Derives the hour value from a Datetime value. Generated hours are expressed according to the 24-hour clock. |
| MINUTE Function | Derives the minutes value from a Datetime value. Minutes are expressed as integers from 0 to 59. |
| SECOND Function | Derives the seconds value from a Datetime value. Source value can be a a reference to a column containing Datetime values or a literal. |

**Source:**

| date |
|---|
| 2/8/16 15:41 |
| 12/30/15 0:00 |
| 4/26/15 7:07 |

**Transformation:**

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | YEAR (date) |

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | MONTH (date) |

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | MONTHNAME (date) |

| Transformation Name | New formula |
|---|---|

| Parameter: Formula type | Single row formula |
|---|---|
| Parameter: Formula | WEEKDAYNAME (date) |

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | DAY (date) |

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | HOUR (date) |

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | MINUTE (date) |

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | SECOND (date) |

**Results:**

> **NOTE:** If the source Datetime value does not contain a valid input for one of these functions, no value is returned. See the `second_date` column below.

| date | year_date | month_date | monthname_date | weekdayname_date | day_date | hour_date | minute_date | secor |
|---|---|---|---|---|---|---|---|---|
| 2/8/16 15:41 | 2016 | 2 | February | Monday | 8 | 15 | 41 | |
| 12/30 /15 0: 00 | 2015 | 12 | December | Wednesday | 30 | 0 | 0 | |
| 4/26 /15 7: 07 | 2015 | 4 | April | Sunday | 26 | 7 | 7 | |

# WEEKNUM Function

Derives the numeric value for the week within the year (`1`, `2`, etc.). Input must be the output of the `DATE` function or a reference to a column containing Datetime values. The output of this function increments on Sunday.

Week 1 of the year is the week that contains January 1.

> **NOTE:** If the source Datetime value does not include a valid input for this function, a missing value is returned.

**Wrangle vs. SQL:** This function is part of Wrangle , a proprietary data transformation language. Wrangle is not SQL. For more information, see *Wrangle Language*.

> **NOTE:** Depending on the locale where the calculation of this function is performed, the maximum number of weeks in a year may be 52 or 53. It's possible that you could see different results in your browser, Trifacta Photon, and other running environments due to locale.

## Basic Usage

**Column reference example:**

```
weeknum(MyDate)
```

**Output:** Returns the numeric week number values derived from the `MyDate` column.

## Syntax and Arguments

```
weeknum(datetime_col)
```

| Argument | Required? | Data Type | Description |
|---|---|---|---|
| datetime_col | Y | datetime | Name of column whose week number values are to be computed |

For more information on syntax standards, see *Language Documentation Syntax Notes*.

**datetime_col**

Name of the column whose week number value is to be computed.

- Missing values for this function in the source data result in missing values in the output.
- Multiple columns and wildcards are not supported.

> **Tip:** You cannot insert constant Datetime values as inputs to this function. However, you can use the following: `WEEKNUM(DATE(2017,12,20))` .

**Usage Notes:**

| Required? | Data Type | Example Value |
|-----------|-----------|---------------|
| Yes | Datetime | `myDate` |

## Examples

> **Tip:** For additional examples, see *Common Tasks*.

### Example - Day of Date functions

This example illustrates how you can apply functions to derive day-of-week values out of a column of Datetime type.

**Functions**:

| Item | Description |
|------|-------------|
| WEEKDAY Function | Derives the numeric value for the day of the week (1, 2, etc.). Input must be a reference to a column containing Datetime values. |
| WEEKNUM Function | Derives the numeric value for the week within the year (1, 2, etc.). Input must be the output of the `DATE` function or a reference to a column containing Datetime values. The output of this function increments on Sunday. |
| DATEFORMAT Function | Formats a specified Datetime set of values according to the specified date format. Source values can be a reference to a column containing Datetime values. |

**Source:**

| myDate |
|--------|
| 10/30/17 |
| 10/31/17 |
| 11/1/17 |
| 11/2/17 |
| 11/3/17 |
| 11/4/17 |
| 11/5/17 |
| 11/6/17 |

**Transformation:**

The following transformation step generates a numeric value for the day of week in a new column:

| Transformation Name | `New formula` |
|---------------------|---------------|
| **Parameter: Formula type** | `Single row formula` |
| **Parameter: Formula** | `WEEKDAY (myDate)` |
| **Parameter: New column name** | `'weekDayNum'` |

The following step generates a full text version of the name of the day of the week:

| Transformation Name | New formula |
|---|---|
| **Parameter: Formula type** | Single row formula |
| **Parameter: Formula** | DATEFORMAT(myDate, 'EEEE') |
| **Parameter: New column name** | 'weekDayNameFull' |

The following step generates a three-letter abbreviation for the name of the day of the week:

| Transformation Name | New formula |
|---|---|
| **Parameter: Formula type** | Single row formula |
| **Parameter: Formula** | DATEFORMAT(myDate, 'EEE') |
| **Parameter: New column name** | 'weekDayNameShort' |

The following step generates the numeric value of the week within the year:

| Transformation Name | New formula |
|---|---|
| **Parameter: Formula type** | Single row formula |
| **Parameter: Formula** | WEEKNUM (myDate) |
| **Parameter: New column name** | 'weekNum' |

**Results:**

| myDate | weekDayNum | weekDayNameFull | weekDayNameShort | weekNum |
|---|---|---|---|---|
| 10/30/17 | 1 | Monday | Mon | 44 |
| 10/31/17 | 2 | Tuesday | Tue | 44 |
| 11/1/17 | 3 | Wednesday | Wed | 44 |
| 11/2/17 | 4 | Thursday | Thu | 44 |
| 11/3/17 | 5 | Friday | Fri | 44 |
| 11/4/17 | 6 | Saturday | Sat | 44 |
| 11/5/17 | 7 | Sunday | Sun | 45 |
| 11/6/17 | 1 | Monday | Mon | 45 |

# WEEKDAY Function

Derives the numeric value for the day of the week (`1`, `2`, etc.). Input must be a reference to a column containing Datetime values.

> **NOTE:** If the source Datetime value does not include a valid input for this function, a missing value is returned.

> **Tip:** You can use the `DATEFORMAT` function to generate text versions of the day of the week. See Examples below.

**Wrangle vs. SQL:** This function is part of Wrangle , a proprietary data transformation language. Wrangle is not SQL. For more information, see *Wrangle Language*.

## Basic Usage

**Column reference example:**

```
weekday(MyDate)
```

**Output:** Returns the numeric weekday values derived from the `MyDate` column.

## Syntax and Arguments

```
weekday(datetime_col)
```

| Argument | Required? | Data Type | Description |
|----------|-----------|-----------|-------------|
| datetime_col | Y | datetime | Name of column whose weekday values are to be computed |

For more information on syntax standards, see *Language Documentation Syntax Notes*.

**datetime_col**

Name of the column whose day-of-week value is to be computed.

- Missing values for this function in the source data result in missing values in the output.
- Multiple columns and wildcards are not supported.

> **Tip:** You cannot insert constant Datetime values as inputs to this function. However, you can use the following: `WEEKDAY(DATE(12,20,2017))`.

**Usage Notes:**

| Required? | Data Type | Example Value |
|-----------|-----------|---------------|
| Yes | Datetime | `myDate` |

# Examples

> **Tip:** For additional examples, see *Common Tasks*.

### Example - Day of Date functions

This example illustrates how you can apply functions to derive day-of-week values out of a column of Datetime type.

**Functions**:

| Item | Description |
|------|-------------|
| WEEKDAY Function | Derives the numeric value for the day of the week (1, 2, etc.). Input must be a reference to a column containing Datetime values. |
| WEEKNUM Function | Derives the numeric value for the week within the year (1, 2, etc.). Input must be the output of the DATE function or a reference to a column containing Datetime values. The output of this function increments on Sunday. |
| DATEFORMAT Function | Formats a specified Datetime set of values according to the specified date format. Source values can be a reference to a column containing Datetime values. |

**Source:**

| myDate |
|--------|
| 10/30/17 |
| 10/31/17 |
| 11/1/17 |
| 11/2/17 |
| 11/3/17 |
| 11/4/17 |
| 11/5/17 |
| 11/6/17 |

**Transformation:**

The following transformation step generates a numeric value for the day of week in a new column:

| Transformation Name | New formula |
|---------------------|-------------|
| **Parameter: Formula type** | Single row formula |
| **Parameter: Formula** | WEEKDAY (myDate) |
| **Parameter: New column name** | 'weekDayNum' |

The following step generates a full text version of the name of the day of the week:

| Transformation Name | New formula |
|---------------------|-------------|

| Parameter: Formula type | Single row formula |
|---|---|
| Parameter: Formula | DATEFORMAT(myDate, 'EEEE') |
| Parameter: New column name | 'weekDayNameFull' |

The following step generates a three-letter abbreviation for the name of the day of the week:

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | DATEFORMAT(myDate, 'EEE') |
| Parameter: New column name | 'weekDayNameShort' |

The following step generates the numeric value of the week within the year:

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | WEEKNUM (myDate) |
| Parameter: New column name | 'weekNum' |

**Results:**

| myDate | weekDayNum | weekDayNameFull | weekDayNameShort | weekNum |
|---|---|---|---|---|
| 10/30/17 | 1 | Monday | Mon | 44 |
| 10/31/17 | 2 | Tuesday | Tue | 44 |
| 11/1/17 | 3 | Wednesday | Wed | 44 |
| 11/2/17 | 4 | Thursday | Thu | 44 |
| 11/3/17 | 5 | Friday | Fri | 44 |
| 11/4/17 | 6 | Saturday | Sat | 44 |
| 11/5/17 | 7 | Sunday | Sun | 45 |
| 11/6/17 | 1 | Monday | Mon | 45 |

# WEEKDAYNAME Function

Derives the full name from a Datetime value of the corresponding weekday as a String. Source value can be a reference to a column containing Datetime values or a literal.

> **NOTE:** If the source Datetime value does not include a valid input for this function, a missing value is returned.

**Wrangle vs. SQL:** This function is part of Wrangle , a proprietary data transformation language. Wrangle is not SQL. For more information, see *Wrangle Language*.

## Basic Usage

```
weekdayname(MyDate)
```

**Output:** Returns the weekday name from the `MyDate` column.

## Syntax and Arguments

```
<span>weekdayname</span><span>(datetime_col)</span>
```

| Argument | Required? | Data Type | Description |
|----------|-----------|-----------|-------------|
| datetime_col | Y | datetime | Name of column whose weekday name values are to be computed |

For more information on syntax standards, see *Language Documentation Syntax Notes*.

**datetime_col**

Name of the column whose weekday name value is to be computed.

- Missing values for this function in the source data result in missing values in the output.
- Multiple columns and wildcards are not supported.

**Usage Notes:**

| Required? | Data Type | Example Value |
|-----------|-----------|---------------|
| Yes | Datetime | `myDate` |

## Examples

> **Tip:** For additional examples, see *Common Tasks*.

**Example - Date element functions**

This example illustrates to how to use date-related functions to derive specific values for a Datetime column type.

**Functions**:

| Item | Description |
|---|---|
| YEAR Function | Derives the four-digit year value from a Datetime value. Source value can be a a reference to a column containing Datetime values or a literal. |
| MONTH Function | Derives the month integer value from a Datetime value. Source value can be a a reference to a column containing Datetime values or a literal. |
| MONTHNAME Function | Derives the full name from a Datetime value of the corresponding month as a String. Source value can be a reference to a column containing Datetime values or a literal. |
| WEEKDAYNAME Function | Derives the full name from a Datetime value of the corresponding weekday as a String. Source value can be a reference to a column containing Datetime values or a literal. |
| DAY Function | Derives the numeric day value from a Datetime value. Source value can be a a reference to a column containing Datetime values or a literal. |
| HOUR Function | Derives the hour value from a Datetime value. Generated hours are expressed according to the 24-hour clock. |
| MINUTE Function | Derives the minutes value from a Datetime value. Minutes are expressed as integers from 0 to 59. |
| SECOND Function | Derives the seconds value from a Datetime value. Source value can be a a reference to a column containing Datetime values or a literal. |

**Source:**

| date |
|---|
| 2/8/16 15:41 |
| 12/30/15 0:00 |
| 4/26/15 7:07 |

**Transformation:**

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | YEAR (date) |

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | MONTH (date) |

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | MONTHNAME (date) |

| Transformation Name | New formula |
|---|---|

| Parameter: Formula type | Single row formula |
|---|---|
| Parameter: Formula | WEEKDAYNAME (date) |

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | DAY (date) |

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | HOUR (date) |

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | MINUTE (date) |

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | SECOND (date) |

**Results:**

> **NOTE:** If the source Datetime value does not contain a valid input for one of these functions, no value is returned. See the `second_date` column below.

| date | year_date | month_date | monthname_date | weekdayname_date | day_date | hour_date | minute_date | secor |
|---|---|---|---|---|---|---|---|---|
| 2/8/16 15:41 | 2016 | 2 | February | Monday | 8 | 15 | 41 | |
| 12/30 /15 0: 00 | 2015 | 12 | December | Wednesday | 30 | 0 | 0 | |
| 4/26 /15 7: 07 | 2015 | 4 | April | Sunday | 26 | 7 | 7 | |

# HOUR Function

Derives the hour value from a Datetime value. Generated hours are expressed according to the 24-hour clock.

- Source value can be a reference to a column containing literal or Datetime values.
- If the source Datetime value does not include a valid input for this function, a missing value is returned.

**Wrangle vs. SQL:** This function is part of Wrangle , a proprietary data transformation language. Wrangle is not SQL. For more information, see *Wrangle Language*.

## Basic Usage

```
hour(MyDate)
```

**Output:** Generates a column of values that retrieve the two-digit hour values from the `MyDate` column.

## Syntax and Arguments

```
hour(datetime_col)
```

| Argument | Required? | Data Type | Description |
|---|---|---|---|
| datetime_col | Y | datetime | Name of column whose hour values are to be computed |

For more information on syntax standards, see *Language Documentation Syntax Notes*.

**datetime_col**

Name of the column whose hour value is to be computed.

- Missing values for this function in the source data result in missing values in the output.
- Invalid or out-of-range source values generate missing values in the output.
- Multiple columns and wildcards are not supported.

**Usage Notes:**

| Required? | Data Type | Example Value |
|---|---|---|
| Yes | Datetime | `myDate` |

## Examples

> **Tip:** For additional examples, see *Common Tasks*.

**Example - Date element functions**

This example illustrates to how to use date-related functions to derive specific values for a Datetime column type.

**Functions**:

| Item | Description |
|---|---|

| | |
|---|---|
| YEAR Function | Derives the four-digit year value from a Datetime value. Source value can be a a reference to a column containing Datetime values or a literal. |
| MONTH Function | Derives the month integer value from a Datetime value. Source value can be a a reference to a column containing Datetime values or a literal. |
| MONTHNAME Function | Derives the full name from a Datetime value of the corresponding month as a String. Source value can be a reference to a column containing Datetime values or a literal. |
| WEEKDAYNAME Function | Derives the full name from a Datetime value of the corresponding weekday as a String. Source value can be a reference to a column containing Datetime values or a literal. |
| DAY Function | Derives the numeric day value from a Datetime value. Source value can be a a reference to a column containing Datetime values or a literal. |
| HOUR Function | Derives the hour value from a Datetime value. Generated hours are expressed according to the 24-hour clock. |
| MINUTE Function | Derives the minutes value from a Datetime value. Minutes are expressed as integers from 0 to 59. |
| SECOND Function | Derives the seconds value from a Datetime value. Source value can be a a reference to a column containing Datetime values or a literal. |

### Source:

| date |
|---|
| 2/8/16 15:41 |
| 12/30/15 0:00 |
| 4/26/15 7:07 |

### Transformation:

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | YEAR (date) |

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | MONTH (date) |

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | MONTHNAME (date) |

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | WEEKDAYNAME (date) |

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | DAY (date) |

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | HOUR (date) |

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | MINUTE (date) |

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | SECOND (date) |

**Results:**

> **NOTE:** If the source Datetime value does not contain a valid input for one of these functions, no value is returned. See the `second_date` column below.

| date | year_date | month_date | monthname_date | weekdayname_date | day_date | hour_date | minute_date | secor |
|---|---|---|---|---|---|---|---|---|
| 2/8/16 15:41 | 2016 | 2 | February | Monday | 8 | 15 | 41 | |
| 12/30 /15 0: 00 | 2015 | 12 | December | Wednesday | 30 | 0 | 0 | |
| 4/26 /15 7: 07 | 2015 | 4 | April | Sunday | 26 | 7 | 7 | |

# MINUTE Function

Derives the minutes value from a Datetime value. Minutes are expressed as integers from 0 to 59.

- Source value can be a reference to a column containing Datetime values or a literal.
- If the source Datetime value does not include a valid input for this function, a missing value is returned.

**Wrangle vs. SQL:** This function is part of Wrangle , a proprietary data transformation language. Wrangle is not SQL. For more information, see *Wrangle Language*.

## Basic Usage

```
minute(MyDate)
```

**Output:** Returns the two-digit hour values from the `MyDate` column.

## Syntax and Arguments

```
minute(datetime_col)
```

| Argument | Required? | Data Type | Description |
|---|---|---|---|
| datetime_col | Y | datetime | Name of column whose minute values are to be computed |

For more information on syntax standards, see *Language Documentation Syntax Notes*.

**datetime_col**

Name of the column whose minute value is to be computed.

- Missing values for this function in the source data result in missing values in the output.
- Invalid or out-of-range source values generate missing values in the output.
- Multiple columns and wildcards are not supported.

**Usage Notes:**

| Required? | Data Type | Example Value |
|---|---|---|
| Yes | Datetime | `myDate` |

## Examples

> **Tip:** For additional examples, see *Common Tasks*.

**Example - Date element functions**

This example illustrates to how to use date-related functions to derive specific values for a Datetime column type.

**Functions**:

| Item | Description |
|---|---|
| YEAR Function | Derives the four-digit year value from a Datetime value. Source value can be a a reference to a column containing Datetime values or a literal. |
| MONTH Function | Derives the month integer value from a Datetime value. Source value can be a a reference to a column containing Datetime values or a literal. |
| MONTHNAME Function | Derives the full name from a Datetime value of the corresponding month as a String. Source value can be a reference to a column containing Datetime values or a literal. |
| WEEKDAYNAME Function | Derives the full name from a Datetime value of the corresponding weekday as a String. Source value can be a reference to a column containing Datetime values or a literal. |
| DAY Function | Derives the numeric day value from a Datetime value. Source value can be a a reference to a column containing Datetime values or a literal. |
| HOUR Function | Derives the hour value from a Datetime value. Generated hours are expressed according to the 24-hour clock. |
| MINUTE Function | Derives the minutes value from a Datetime value. Minutes are expressed as integers from 0 to 59. |
| SECOND Function | Derives the seconds value from a Datetime value. Source value can be a a reference to a column containing Datetime values or a literal. |

**Source:**

| date |
|---|
| 2/8/16 15:41 |
| 12/30/15 0:00 |
| 4/26/15 7:07 |

**Transformation:**

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | YEAR (date) |

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | MONTH (date) |

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | MONTHNAME (date) |

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |

| Parameter: Formula | `WEEKDAYNAME (date)` |
|---|---|

| Transformation Name | `New formula` |
|---|---|
| **Parameter: Formula type** | `Single row formula` |
| **Parameter: Formula** | `DAY (date)` |

| Transformation Name | `New formula` |
|---|---|
| **Parameter: Formula type** | `Single row formula` |
| **Parameter: Formula** | `HOUR (date)` |

| Transformation Name | `New formula` |
|---|---|
| **Parameter: Formula type** | `Single row formula` |
| **Parameter: Formula** | `MINUTE (date)` |

| Transformation Name | `New formula` |
|---|---|
| **Parameter: Formula type** | `Single row formula` |
| **Parameter: Formula** | `SECOND (date)` |

**Results:**

> **NOTE:** If the source Datetime value does not contain a valid input for one of these functions, no value is returned. See the `second_date` column below.

| date | year_date | month_date | monthname_date | weekdayname_date | day_date | hour_date | minute_date | secon |
|---|---|---|---|---|---|---|---|---|
| 2/8/16 15:41 | 2016 | 2 | February | Monday | 8 | 15 | 41 | |
| 12/30 /15 0: 00 | 2015 | 12 | December | Wednesday | 30 | 0 | 0 | |
| 4/26 /15 7: 07 | 2015 | 4 | April | Sunday | 26 | 7 | 7 | |

# SECOND Function

Derives the seconds value from a Datetime value. Source value can be a a reference to a column containing Datetime values or a literal.

- If the source Datetime value does not include a valid input for this function, a missing value is returned.
- If the input values do not contain milliseconds, the generated output is expressed as integers from 0 to 59.
- If the input values contain milliseconds, the generated output is a floating point value.

**Wrangle vs. SQL:** This function is part of Wrangle , a proprietary data transformation language. Wrangle is not SQL. For more information, see *Wrangle Language*.

## Basic Usage

```
second(MyDate)
```

**Output:** Generates a column of values that retrieve the two-digit hour values from the `MyDate` column.

## Syntax and Arguments

```
second(datetime_col)
```

| Argument | Required? | Data Type | Description |
|----------|-----------|-----------|-------------|
| datetime_col | Y | datetime | Name of column whose second values are to be computed |

For more information on syntax standards, see *Language Documentation Syntax Notes*.

**datetime_col**

Name of the column whose seconds value is to be computed.

- Missing values for this function in the source data result in missing values in the output.
- Invalid or out-of-range source values generate missing values in the output.
- Multiple columns and wildcards are not supported.

**Usage Notes:**

| Required? | Data Type | Example Value |
|-----------|-----------|---------------|
| Yes | Datetime | `myDate` |

## Examples

> **Tip:** For additional examples, see *Common Tasks*.

**Example - Date element functions**

This example illustrates to how to use date-related functions to derive specific values for a Datetime column type.

**Functions**:

| Item | Description |
|---|---|
| YEAR Function | Derives the four-digit year value from a Datetime value. Source value can be a a reference to a column containing Datetime values or a literal. |
| MONTH Function | Derives the month integer value from a Datetime value. Source value can be a a reference to a column containing Datetime values or a literal. |
| MONTHNAME Function | Derives the full name from a Datetime value of the corresponding month as a String. Source value can be a reference to a column containing Datetime values or a literal. |
| WEEKDAYNAME Function | Derives the full name from a Datetime value of the corresponding weekday as a String. Source value can be a reference to a column containing Datetime values or a literal. |
| DAY Function | Derives the numeric day value from a Datetime value. Source value can be a a reference to a column containing Datetime values or a literal. |
| HOUR Function | Derives the hour value from a Datetime value. Generated hours are expressed according to the 24-hour clock. |
| MINUTE Function | Derives the minutes value from a Datetime value. Minutes are expressed as integers from 0 to 59. |
| SECOND Function | Derives the seconds value from a Datetime value. Source value can be a a reference to a column containing Datetime values or a literal. |

**Source:**

| date |
|---|
| 2/8/16 15:41 |
| 12/30/15 0:00 |
| 4/26/15 7:07 |

**Transformation:**

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | YEAR (date) |

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | MONTH (date) |

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | MONTHNAME (date) |

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |

| Parameter: Formula | WEEKDAYNAME (date) |
|---|---|

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | DAY (date) |

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | HOUR (date) |

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | MINUTE (date) |

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | SECOND (date) |

**Results:**

> **NOTE:** If the source Datetime value does not contain a valid input for one of these functions, no value is returned. See the `second_date` column below.

| date | year_date | month_date | monthname_date | weekdayname_date | day_date | hour_date | minute_date | secon |
|---|---|---|---|---|---|---|---|---|
| 2/8/16 15:41 | 2016 | 2 | February | Monday | 8 | 15 | 41 | |
| 12/30 /15 0: 00 | 2015 | 12 | December | Wednesday | 30 | 0 | 0 | |
| 4/26 /15 7: 07 | 2015 | 4 | April | Sunday | 26 | 7 | 7 | |

# UNIXTIME Function

Derives the Unixtime (or epoch time) value from a Datetime value. Source value can be a reference to a column containing Datetime values.

**Unix time** is a date-time format used to express the number of milliseconds that have elapsed since January 1, 1970 00:00:00 (UTC).

- Unix time does not handle the extra seconds that occur on the extra day of leap years.

This function factors any timezone values in the inputs.

- If you have a column with multiple time zones, you can convert the column to Unixtime so you can perform Date/Time operations with a standardized time zone.
- If you want to work with local times, you can truncate the time zone or use other Date functions.
- If the source Datetime value does not include a valid input for this function, a missing value is returned.

**Wrangle vs. SQL:** This function is part of Wrangle , a proprietary data transformation language. Wrangle is not SQL. For more information, see *Wrangle Language*.

## Basic Usage

```
unixtime(MyDate)
```

**Output:** Returns the Unix time values from the `MyDate` column.

## Syntax and Arguments

```
unixtime(datetime_col)
```

| Argument | Required? | Data Type | Description |
|---|---|---|---|
| datetime_col | Y | datetime | Name of column whose Unix time values are to be computed |

For more information on syntax standards, see *Language Documentation Syntax Notes*.

**datetime_col**

Name of the column whose Unix time value is to be computed.

- Missing values for this function in the source data result in missing values in the output.
- Invalid or out-of-range source values generate missing values in the output.
- Multiple columns and wildcards are not supported.
- Includes time zone offset when it converts Date/Time values to unixtime.
- If the date value does not include a time zone, unixtime uses UTC (0:00).

**Usage Notes:**

| Required? | Data Type | Example Value |
|---|---|---|
| Yes | Datetime | `myDate` |

# Examples

> **Tip:** For additional examples, see *Common Tasks*.

### Example - Unix time generation and formatting

This example illustrates how you can use functions to manipulate Unix time values in a column of Datetime type.

**Functions**:

| Item | Description |
|------|-------------|
| UNIXTIME Function | Derives the Unixtime (or epoch time) value from a Datetime value. Source value can be a reference to a column containing Datetime values. |

**Source:**

| date |
|------|
| 2/8/16 15:41 |
| 12/30/15 0:00 |
| 4/26/15 7:07 |

### Transformation:

Use the following transformation step to generate a column containing the above values as Unix timecode values:

| | |
|---|---|
| **Transformation Name** | `New formula` |
| **Parameter: Formula type** | `Single row formula` |
| **Parameter: Formula** | `UNIXTIME (date)` |
| **Parameter: New column name** | `'unixtime_date'` |

**Results:**

> **NOTE:** If the source Datetime value does not contain a valid input for one of these functions, no value is returned.

| date | unixtime_date |
|------|---------------|
| 2/8/16 15:41 | 1454946120000 |
| 12/30/15 0:00 | 1451433600000 |
| 4/26/15 7:07 | 1430032020000 |

# NOW Function

Derives the timestamp for the current time in UTC time zone. You can specify a different time zone by optional parameter.

For this function, the values that you see in the Transformer grid are generated during the preview. These values will differ from the values that are generated later, when the job is executed.

> **NOTE:** Some Datetime functions do not allow the nesting of `NOW` and `TODAY` functions. You should create a separate column containing these values.

Other differences:

- If you refresh the page for the Transformer grid, the function is recalculated.
- If you re-open the dataset in the Transformer page, the function is recalculated.
- If you have multiple versions of the function in the same dataset, you are likely to end up with different generated timestamps. The difference in their values cannot be accurately predicted.

> **Tip:** If you wish to use the same computed value for this function across your dataset, you should generate a column containing values for the function and then base all of your other calculations off of these column values.

> **NOTE:** If no time zone is specified, the default is UTC time zone. Time values are returned in 24-hour time.

For more information on generating the date value only for today, see *TODAY Function*.

**Wrangle vs. SQL:** This function is part of Wrangle , a proprietary data transformation language. Wrangle is not SQL. For more information, see *Wrangle Language*.

## Basic Usage

**Basic Example:**

```
now()
```

**Output:** Returns the current timestamp in UTC time zone.

**Example with Time Zone:**

```
<span>now(&apos;America/New York&apos;)</span>
```

**Output:** Returns the current timestamp based on the time in the Eastern U.S. time zone.

## Syntax and Arguments

```
now(str_timezone)
```

| Argument | Required? | Data Type | Description |
|---|---|---|---|
| str_timezone | N | string | String value for the time zone for which the timestamp is calculated. |

For more information on syntax standards, see *Language Documentation Syntax Notes*.

**str_timezone**

String value for the time zone to use.

**Usage Notes:**

| Required? | Data Type | Example Value |
|-----------|-----------|---------------|
| No | String | `'America/New York'` |

## Examples

> **Tip:** For additional examples, see *Common Tasks*.

**Example - Flight Status report**

This example illustrates you to generate the date and time values for the current date and timestamp in the specified time zone.

**Functions**:

| Item | Description |
|------|-------------|
| NOW Function | Derives the timestamp for the current time in UTC time zone. You can specify a different time zone by optional parameter. |
| TODAY Function | Derives the value for the current date in UTC time zone. You can specify a different time zone by optional parameter. |
| DATEDIF Function | Calculates the difference between two valid date values for the specified units of measure. |

**Source:**

The following table includes flight arrival information for Los Angeles International airport.

| FlightNumber | Gate | Arrival |
|--------------|------|---------|
| 1234 | 1 | 2/15/17 11:35 |
| 212 | 2 | 2/15/17 11:58 |
| 510 | 3 | 2/15/17 11:21 |
| 8401 | 4 | 2/15/17 12:08 |
| 99 | 5 | 2/16/17 12:12 |
| 116 | 6 | 2/16/17 13:32 |
| 876 | 7 | 2/15/17 16:43 |
| 9494 | 8 | 2/15/17 21:00 |
| 102 | 9 | 2/14/17 19:21 |
| 77 | 10 | 2/16/17 12:31 |

**Transformation:**

You are interested in generating a status report on today's flights. To assist, you must generate columns with the current date and time values:

> **Tip:** You should create separate columns containing static values for `NOW` and `TODAY` functions. Avoid creating multiple instances of each function in your dataset, as the values calculated in them can vary at execution time.

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | NOW('America\/Los_Angeles') |
| Parameter: New column name | 'currentTime' |

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | TODAY('America\/Los_Angeles') |
| Parameter: New column name | 'currentDate' |

Next, you want to identify the flights that are landing today. In this case, you can use the `DATEDIF` function to determine if the `Arrival` value matches the `currentTime` value within one day:

> **NOTE:** The `DATEDIF` function computes difference based on the difference from the first date to the second date based on the unit of measure. So, a timestamp that is 23 hours difference from the base timestamp can be within the same unit of day, even though the dates may be different (`2/15/2017` vs. `2/14/2017`).

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | DATEDIF(currentDate, Arrival, day) |
| Parameter: New column name | 'today' |

Since you are focusing on today only, you can remove all of the rows that do not apply to today:

| Transformation Name | Filter rows |
|---|---|
| Parameter: Condition | Custom formula |
| Parameter: Type of formula | Custom single |
| Parameter: Condition | today <> 0 |
| Parameter: Action | Delete matching rows |

Now focusing on today's dates, you can calculate the difference between the current time and the arrival time by the minute:

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | DATEDIF(currentTime, Arrival, minute) |
| Parameter: New column name | 'status' |

Using the numeric values in the `status` column, you can compose the following transform, which identifies status of each flight:

| Transformation Name | Edit column with formula |
|---|---|
| Parameter: Columns | status |
| Parameter: Formula | if(status < -20, 'arrived', if(status > 20, 'scheduled', if (status <= 0, 'landed', 'arriving'))) |

**Results:**

You now have a daily flight status report:

| currentDate | currentTime | FlightNumber | Gate | Arrival | status | today |
|---|---|---|---|---|---|---|
| 2017-02-15 | 2017-02-15·11:46:12 | 1234 | 1 | 2/15/17 11:35 | landed | 0 |
| 2017-02-15 | 2017-02-15·11:46:12 | 212 | 2 | 2/15/17 11:58 | arriving | 0 |
| 2017-02-15 | 2017-02-15·11:46:12 | 510 | 3 | 2/15/17 11:21 | arrived | 0 |
| 2017-02-15 | 2017-02-15·11:46:12 | 8401 | 4 | 2/15/17 12:08 | scheduled | 0 |
| 2017-02-15 | 2017-02-15·11:46:12 | 876 | 7 | 2/15/17 16:43 | scheduled | 0 |
| 2017-02-15 | 2017-02-15·11:46:12 | 9494 | 8 | 2/15/17 21:00 | scheduled | 0 |
| 2017-02-15 | 2017-02-15·11:46:12 | 102 | 9 | 2/14/17 19:21 | arrived | 0 |

The `currentDate`, `currentTime`, and `today` columns can be deleted.

# TODAY Function

Derives the value for the current date in UTC time zone. You can specify a different time zone by optional parameter.

For this function, the values that you see in the Transformer grid are generated during the preview. These values will differ from the values that are generated later, when the job is executed.

> **NOTE:** Some Datetime functions do not allow the nesting of NOW and TODAY functions. You should create a separate column containing these values.

Other differences:

- If you refresh the page for the Transformer grid, the function is recalculated.
- If you re-open the dataset in the Transformer page, the function is recalculated.
- If you have multiple versions of the function in the same dataset, you are likely to end up with different generated timestamps. The difference in their values cannot be accurately predicted.

> **Tip:** If you wish to use the same computed value for this function across your dataset, you should generate a column containing values for the function and then base all of your other calculations off of these column values.

> **NOTE:** If no time zone is specified, the default is UTC time zone.

For more information on generating the date and time stamp value for the current time, see *NOW Function*.

**Wrangle vs. SQL:** This function is part of Wrangle , a proprietary data transformation language. Wrangle is not SQL. For more information, see *Wrangle Language*.

## Basic Usage

**Basic Example:**

```
today()
```

**Output:** Returns the Datetime value for the current date in UTC time zone.

**Example with Time Zone:**

```
today(&apos;America/New York&apos;)
```

**Output:** Returns the Datetime value for the current date based on the time in the Eastern U.S. time zone.

## Syntax and Arguments

```
today([str_timezone])
```

| Argument | Required? | Data Type | Description |
|---|---|---|---|
| str_timezone | N | string | String value for the time zone for which the date value is calculated. |

For more information on syntax standards, see *Language Documentation Syntax Notes.*

**str_timezone**

String value for the time zone to use.

**Usage Notes:**

| Required? | Data Type | Example Value |
|---|---|---|
| No | String | `'America/New York'` |

## Examples

> **Tip:** For additional examples, see *Common Tasks.*

**Example - Flight Status report**

This example illustrates you to generate the date and time values for the current date and timestamp in the specified time zone.

**Functions**:

| Item | Description |
|---|---|
| NOW Function | Derives the timestamp for the current time in UTC time zone. You can specify a different time zone by optional parameter. |
| TODAY Function | Derives the value for the current date in UTC time zone. You can specify a different time zone by optional parameter. |
| DATEDIF Function | Calculates the difference between two valid date values for the specified units of measure. |

**Source:**

The following table includes flight arrival information for Los Angeles International airport.

| FlightNumber | Gate | Arrival |
|---|---|---|
| 1234 | 1 | 2/15/17 11:35 |
| 212 | 2 | 2/15/17 11:58 |
| 510 | 3 | 2/15/17 11:21 |
| 8401 | 4 | 2/15/17 12:08 |
| 99 | 5 | 2/16/17 12:12 |
| 116 | 6 | 2/16/17 13:32 |
| 876 | 7 | 2/15/17 16:43 |
| 9494 | 8 | 2/15/17 21:00 |
| 102 | 9 | 2/14/17 19:21 |
| 77 | 10 | 2/16/17 12:31 |

**Transformation:**

You are interested in generating a status report on today's flights. To assist, you must generate columns with the current date and time values:

> **Tip:** You should create separate columns containing static values for `NOW` and `TODAY` functions. Avoid creating multiple instances of each function in your dataset, as the values calculated in them can vary at execution time.

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | NOW('America\/Los_Angeles') |
| Parameter: New column name | 'currentTime' |

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | TODAY('America\/Los_Angeles') |
| Parameter: New column name | 'currentDate' |

Next, you want to identify the flights that are landing today. In this case, you can use the `DATEDIF` function to determine if the `Arrival` value matches the `currentTime` value within one day:

> **NOTE:** The `DATEDIF` function computes difference based on the difference from the first date to the second date based on the unit of measure. So, a timestamp that is 23 hours difference from the base timestamp can be within the same unit of day, even though the dates may be different (`2/15/2017` vs. `2/14/2017`).

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | DATEDIF(currentDate, Arrival, day) |
| Parameter: New column name | 'today' |

Since you are focusing on today only, you can remove all of the rows that do not apply to today:

| Transformation Name | Filter rows |
|---|---|
| Parameter: Condition | Custom formula |
| Parameter: Type of formula | Custom single |
| Parameter: Condition | today <> 0 |
| Parameter: Action | Delete matching rows |

Now focusing on today's dates, you can calculate the difference between the current time and the arrival time by the minute:

| Transformation Name | `New formula` |
|---|---|
| **Parameter: Formula type** | `Single row formula` |
| **Parameter: Formula** | `DATEDIF(currentTime, Arrival, minute)` |
| **Parameter: New column name** | `'status'` |

Using the numeric values in the `status` column, you can compose the following transform, which identifies status of each flight:

| Transformation Name | `Edit column with formula` |
|---|---|
| **Parameter: Columns** | `status` |
| **Parameter: Formula** | `if(status < -20, 'arrived', if(status > 20, 'scheduled', if (status <= 0, 'landed', 'arriving')))` |

**Results:**

You now have a daily flight status report:

| currentDate | currentTime | FlightNumber | Gate | Arrival | status | today |
|---|---|---|---|---|---|---|
| 2017-02-15 | 2017-02-15·11:46:12 | 1234 | 1 | 2/15/17 11:35 | landed | 0 |
| 2017-02-15 | 2017-02-15·11:46:12 | 212 | 2 | 2/15/17 11:58 | arriving | 0 |
| 2017-02-15 | 2017-02-15·11:46:12 | 510 | 3 | 2/15/17 11:21 | arrived | 0 |
| 2017-02-15 | 2017-02-15·11:46:12 | 8401 | 4 | 2/15/17 12:08 | scheduled | 0 |
| 2017-02-15 | 2017-02-15·11:46:12 | 876 | 7 | 2/15/17 16:43 | scheduled | 0 |
| 2017-02-15 | 2017-02-15·11:46:12 | 9494 | 8 | 2/15/17 21:00 | scheduled | 0 |
| 2017-02-15 | 2017-02-15·11:46:12 | 102 | 9 | 2/14/17 19:21 | arrived | 0 |

The `currentDate`, `currentTime`, and `today` columns can be deleted.

# PARSEDATE Function

Evaluates an input against the default input formats or (if specified) an array of Datetime format strings in their listed order. If the input matches one of the formats, the function outputs a Datetime value.

- Inputs can be of any type.
- If the input is not a Datetime value and does match one of the specified formats, the output is in the following format: `yyyy-MM-dd HH:mm:ss`.
- If the input is a Datetime value and does match, the output is in the input's Datetime format.

> **NOTE:** Parsing of dates may depend on how dates are recognized by the running environment where the transformation is executed. In some cases, locale settings for the backend running environment may be different from your user locale settings, which affect how values are interpreted in the Transformer page. If you are having difficulties with execution of the PARSEDATE function, you can try to set your locale settings to U.S. to fix your recipes. This is a known issue.

After you have converted your strings values to dates, if a sufficient percentage of input strings from a column are successfully converted to one of the matching formats, the column may be retyped as Datetime.

- Designer Cloud Powered by Trifacta® Enterprise Edition supports a wide variety of formats for Datetime fields.
- You can explore the available Datetime formats through the Transformer page. From a column's type drop-down, select **Date/Time**. Then, select the formatting category. From the displayed drop-down, you can select a specific format. When this transform step is added to your recipe, you can edit it to see how the format is specified in Wrangle .
- You can then use the DATEFORMAT function to convert the output values to your preferred Datetime format. See *DATEFORMAT Function*.

**Wrangle vs. SQL:** This function is part of Wrangle , a proprietary data transformation language. Wrangle is not SQL. For more information, see *Wrangle Language*.

## Basic Usage

```
parsedate(strDate, ['yyyy-MM-dd','yyyy-MM','yyyy/MM','yyyy-MM-dd'])
```

**Output:** Returns a value structured in `yyyy-MM-dd HH:mm:ss` format if the input value in `strDate` matches any of default formats, which are the following:

```
'yyyy-MM-dd HH:mm:ss'
'yyyy/MM/dd HH:mm:ss'
'yyyy-MM-dd'
'yyyy/MM/dd'
```

```
parsedate(strDate, ['yyyy-MM','yyyy/MM',])
```

**Output:** Returns a value structured in `yyyy-MM-dd HH:mm:ss` format if the input value in `strDate` matches any of the listed formats for dates.

## Syntax and Arguments

```
parsedate(date_col, date_formats_array)
```

| Argument | Required? | Data Type | Description |
|---|---|---|---|
| date_col | Y | any | Literal, name of a column, or a function returning values to match |
| date_formats_array | N | string | (optional) An array of date format strings that are used to match against input values. |

For more information on syntax standards, see *Language Documentation Syntax Notes*.

**date_col**

Literal, column name, or function returning values that are to be evaluated for conversion to Datetime values.

- Inputs values can be of any type.
- Missing values for this function in the source data result in null values in the output.
- Multiple columns and wildcards are not supported.

**Usage Notes:**

| Required? | Data Type | Example Value |
|---|---|---|
| Yes | any | `'February 24, 2019'` |

**date_formats_array**

Array of String values of the date formats to evaluate the inputs.

- When a non-Datetime input value matches one of the date formats in the array, the output is the input value converted to the following format:

```
yyyy-MM-dd HH:mm:ss
```

- Datetime inputs are outputted in their source format.

Designer Cloud Powered by Trifacta Enterprise Edition supports Java formatting strings, with some exceptions.

> **NOTE:** If the platform cannot recognize the date format string, a null value is written as the output.

**Usage Notes:**

| Required? | Data Type | Example Value |
|---|---|---|
| No | Array of Strings | `[ 'yyyy-MM' , 'yyyy/MM' ]` |

## Examples

> **Tip:** For additional examples, see *Common Tasks*.

## Example - formatting date values

This example illustrates several ways of wrangling heterogeneous date values, including the use of the `DATEFORMAT` function.

### Source:

Your dataset includes the following messy date values:

| MyDateStrings |
|---|
| 2/1/00 14:20 |
| 4/5/10 11:25 |
| 6/7/99 22:00 |
| 13/7/1999 22:00 |
| 12-20-1894 15:45:00 |
| 08-12-1956 22:01:04 |

### Transformation:

To enable easier comparison in the data grid, you choose to create a new column with the parsed values. From the above, you identify two date formats:

```
'MM/dd/yy hh:mm'
'MM/dd/YYYY hh:mm:ss'
```

> **NOTE:** Since only one of the above formats matches the default formats, you must specify both in the transformation to perform the proper evalation.

You create the following transformation to parse them:

| | |
|---|---|
| **Transformation Name** | `New formula` |
| **Parameter: Formula type** | `Single row formula` |
| **Parameter: Formula** | `parsedate(myDateStrings, ['MM/dd/yy hh:mm','MM/dd/YYYY hh:mm:ss'])` |
| **Parameter: New column name** | `'myParsedDates'` |

When the above step is added to the recipe, the output is as follows:

| MyDateStrings | myParsedDates |
|---|---|
| 2/1/00 14:20 | 2000-02-01 14:20:00 |
| 4/5/10 11:25 | 2010-04-05 11:25:00 |
| 6/7/99 22:00 | 1999-06-07 22:00:00 |
| 13/7/1999 22:00 | 13/7/1999 22:00 |
| 12-20-1894 15:45:00 | 1894-12-20 15:45:00 |

| | |
|---|---|
| 08-12-1956 22:01:04 | 1956-08-12 22:01:04 |

The output `myParsedDates` column is retyped as a Datetime column with one mismatched value: `3/7/1999 22:00`.

This value does not match any of our date formats specified in the array. The solution is to modify the recipe step to include the appropriate format as part of the array:

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | `parsedate(myDateStrings, ['MM/dd/yy hh:mm','MM/dd/YYYY hh:mm:ss','dd/MM/yyyy hh:mm'])` |
| Parameter: New column name | `'myParsedDates'` |

### Results:

| MyDateStrings | myParsedDates |
|---|---|
| 2/1/00 14:20 | 2000-02-01 14:20:00 |
| 4/5/10 11:25 | 2010-04-05 11:25:00 |
| 6/7/99 22:00 | 1999-06-07 22:00:00 |
| 13/7/1999 22:00 | 1999-07-13 22:00:00 |
| 12-20-1894 15:45:00 | 1894-12-20 15:45:00 |
| 08-12-1956 22:01:04 | 1956-08-12 22:01:04 |

## Example - type parsing functions

This example shows how to use parsing functions for evaluating input values against the function-specific data type.

### Functions:

| Item | Description |
|---|---|
| PARSEBOOL Function | Evaluates a String input against the Boolean datatype. If the input matches, the function outputs a Boolean value. Input can be a literal, a column of values, or a function returning String values. |
| PARSEDATE Function | Evaluates an input against the default input formats or (if specified) an array of Datetime format strings in their listed order. If the input matches one of the formats, the function outputs a Datetime value. |
| PARSEFLOAT Function | Evaluates a String input against the Decimal datatype. If the input matches, the function outputs a Decimal value. Input can be a literal, a column of values, or a function returning String values. |
| PARSEINT Function | Evaluates a String input against the Integer datatype. If the input matches, the function outputs an Integer value. Input can be a literal, a column of values, or a function returning String values. |

### Source:

The following table contains data on a series of races.

| raceId | disqualified | date | racerId | time_sc |
|---|---|---|---|---|
| 1 | FALSE | 2/1/20 | 1 | 24.22 |

| 2 | f | 2/8/20 | 1 | 25 |
|---|---|---|---|---|
| 3 | no | 2/8/20 | 1 | 24.11 |
| 4 | n | 1-Feb-20 | 2 | 26.1 |
| 5 | TRUE | 8-Feb-20 | 2.2 | -25.22 |
| 6 | t | 2/8/2020  10:16:00 AM | 2 | 25.44 |
| 7 | yes | 2/1/20 | 3 | 24 |
| 8 | y | 2/8/20 | 33 | 29.22 |
| 9 | 0 | 2/8/20 | 3 | 24.78 |
| 10 | 1 | 1-Feb-20 | 4 | 26.2.1 |
| 11 | FALSE | 8-Feb-20 | | 28.22 sec |
| 12 | FALSE | 2/8/2020  10:16:00 AM | 4 | 27.11 |

As you can see, this dataset has variation in values (FALSE, f, no, n) and problems with the data.

**Transformation:**

When the data is first imported, it may be properly typed for each column. To use the parsing functions, these columns should be converted to String data type:

| Transformation Name | Change column data type |
|---|---|
| Parameter: Columns | disqualified,date,racerId,time_sc |
| Parameter: New type | String |

Now, you can parse individual columns.

disqualified column:

| Transformation Name | Edit column with formula |
|---|---|
| Parameter: Columns | disqualified |
| Parameter: Formula | PARSEBOOL($col) |

racerId column:

| Transformation Name | Edit column with formula |
|---|---|
| Parameter: Columns | racerId |
| Parameter: Formula | PARSEINT($col) |

time_sc column:

| Transformation Name | Edit column with formula |
|---|---|
| Parameter: Columns | time_sc |
| Parameter: Formula | PARSEFLOAT($col) |

date column:

For the date column, the PARSEDATE function supports a default set of Datetime formats. Since some of the listed formats are different from these defaults, you must specify all of the formats. These formats are specified as an array of string values as the second argument of the function:

> **Tip:** For the PARSEDATE function, it's useful to use the Preview to verify that all of the dates in the column are represented in the array of output formats. You can see the available output formats through the data type menu at the top of a column in the Transformer Page.

| Transformation Name | `Edit column with formula` |
|---|---|
| Parameter: Columns | `date` |
| Parameter: Formula | `PARSEDATE($col, ['yyyy-MM-dd','yyyy\/MM\/dd','M\/d\/yyy hh:mm','MMMM d, yyyy','MMM d, yyyy'])` |

After all of the date values have been standardized to the output format of the PARSEDATE function, you may choose to remove the time element of the values:

| Transformation Name | `Replace text or pattern` |
|---|---|
| Parameter: Column | `date` |
| Parameter: Find | `` `{digit}{2}:{digit}{2}:{digit}{2}{end}` `` |
| Parameter: Replace with | `''` |

**Results:**

After executing the above steps, the data appears as follows. Notes on each column's output are below the table.

| raceId | disqualified | date | racerId | time_sc |
|---|---|---|---|---|
| 1 | false | 2020-02-01 | 1 | 24.22 |
| 2 | false | 2020-02-08 | 1 | 25 |
| 3 | false | 2020-02-08 | 1 | 24.11 |
| 4 | false | 2020-02-01 | 2 | 26.1 |
| 5 | true | 2020-02-08 | *null* | -25.22 |
| 6 | true | 2020-02-08 | 2 | 25.44 |
| 7 | true | 2020-02-01 | 3 | 24 |
| 8 | true | 2020-02-08 | 33 | 29.22 |
| 9 | false | 2020-02-08 | 3 | 24.78 |
| 10 | true | 2020-02-01 | 4 | *null* |
| 11 | false | 2020-02-08 | *null* | *null* |
| 12 | false | 2020-02-08 | 4 | 27.11 |

disqualified column:

- The PARSEBOOL function normalizes all valid Boolean values to either `false` or `true`.

racerId column:

- The PARSEINT function writes invalid values as null values.
- The function writes empty values as null values.
- The value `33` remains, since it is a valid Integer. This value should be fixed manually.

time_sc:

- The PARSEFLOAT function writes the source value `25.00` as `25` in output.
- The source value `-25.22` remains. However, since this is time-based data, it needs to be fixed.
- Invalid values are written as nulls.

date column:

- All values are written in the standardized format: `yyyy-MM-dd HH:mm:ss`. Time data has been stripped.

# NETWORKDAYS Function

**Contents:**

Calculates the number of working days between two specified dates, assuming Monday - Friday workweek. Optional list of holidays can be specified.

- Inputs can be column references or the outputs of the DATE or TIME functions.
    - See *DATE Function*.
    - See *TIME Function*.
- The first value is used as the baseline to compare the date values.
- If the first date value occurs after the second date value, a null value is returned.

**Wrangle vs. SQL:** This function is part of Wrangle , a proprietary data transformation language. Wrangle is not SQL. For more information, see *Wrangle Language*.

## Basic Usage

```
networkdays(StartDate, EndDate)
```

**Output:** Returns the number of working days between `StartDate` and `EndDate`.

## Syntax and Arguments

```
networkdays(date1,date2,[array_holiday])
```

| Argument | Required? | Data Type | Description |
|---|---|---|---|
| date1 | Y | datetime | Starting date to compare |
| date2 | Y | datetime | Ending date to compare |
| array_holiday | N | array | An array of string values representing the valid dates of holidays. |

For more information on syntax standards, see *Language Documentation Syntax Notes*.

**date1, date2**

Date values can be column references or output of the DATE function or the TIME function.

- For more information, see *DATE Function*.
- For more information, see *TIME Function*.

Date values to compared in working days.

- If `date2` > `date1`, then results are positive.

- If `date2` < `date1`, then a null value is returned.

If `date1` and `date2` have a specified time zone offset, the function calculates the difference including the timezone offsets.

- If `date1` does not have a specified time zone but `date2` does, the function uses the local time in the same time zone as `date2` to calculate the difference. The functions returns the difference without the time zone offset.

**Usage Notes:**

| Required? | Data Type | Example Value |
|---|---|---|
| Yes | Datetime (Column reference or date output of DATE or TIME function) | `LastContactDate` |

**array_holiday**

An array containing the list of holidays, which are factored in the calculation of working days.

Values in the array must be in either of the following formats:

```
['2020-12-24','2020-12-25']
['2020/12/24','2020/12/25']
```

**Usage Notes:**

| Required? | Data Type | Example Value |
|---|---|---|
| Yes | Array | ['2018-12,24','2018-12-25','2018-12-31','2019-01-01'] |

## Examples

> **Tip:** For additional examples, see *Common Tasks*.

**Example - Date diffing functions**

This example demonstrates how to calculate the number of days between two input dates.

**Functions**:

| Item | Description |
|---|---|
| DATEDIF Function | Calculates the difference between two valid date values for the specified units of measure. |
| NETWORKDAYS Function | Calculates the number of working days between two specified dates, assuming Monday - Friday workweek. Optional list of holidays can be specified. |
| NETWORKDAYS INTL Function | Calculates the number of working days between two specified dates. Optionally, you can specify which days of the week are working days as an input parameter. Optional list of holidays can be specified. |
| WORKDAY Function | Calculates the work date that is before or after a start date, as specified by a number of days. A set of holiday dates can be optionally specified. |

| WORKDAYINTL Function | Calculates the work date that is before or after a start date, as specified by a number of days. You can also specify which days of the week are working days and a list of holidays via parameters. |
|---|---|

**Source:**

The following dataset contains two columns of dates.

- The first column values are constant. This date falls on a Monday.

| Date1 | Date2 |
|---|---|
| 2020-03-09 | 2020-03-13 |
| 2020-03-09 | 2020-03-06 |
| 2020-03-09 | 2020-03-16 |
| 2020-03-09 | 2020-03-23 |
| 2020-03-09 | 2020-04-10 |
| 2020-03-09 | 2021-03-10 |

**Transformation:**

The first transformation calculates the number of raw days between the two dates:

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | datedif(Date1, Date2, day) |
| Parameter: New column name | 'datedif' |

This step computes the number of working days between the two dates. Assumptions:

- Workweek is Monday - Friday.
- There are no holidays.

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | networkdays(Date1, Date2, []) |
| Parameter: New column name | 'networkDays' |

For some, March 17 is an important date, especially if you are Irish. To add St. Patrick's Day to the list of holidays, you could add the following transformation:

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | networkdays(Date1, Date2, ['2020-03-17']) |
| Parameter: New column name | 'networkDaysStPatricks' |

In the following transformation, the NETWORKDAYSINTL function is applied so that you can specify the working days in the week:

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | networkdaysintl(Date1, Date2, '1000011', []) |
| Parameter: New column name | 'networkDaysIntl' |

The following two functions enable you to calculate a specific working date based on an input date and integer number of days before or after it. In the following, the date that is five working days before the `Date2` column is computed:

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | workday(Date2, -5) |
| Parameter: New column name | 'workday' |

Suppose you wish to factor in a four-day workweek, in which Friday through Sunday is considered the weekend:

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | workdayintl(Date2, -5,'0000111') |
| Parameter: New column name | 'workdayintl' |

**Results:**

| Date1 | Date2 | workdayintl | workday | networkDaysIntl | networkDaysStPatricks | networkDays | datedif |
|---|---|---|---|---|---|---|---|
| 2020-03-09 | 2020-03-13 | 2020-03-05 | 2020-03-06 | 4 | 5 | 5 | 4 |
| 2020-03-09 | 2020-03-06 | 2020-02-27 | 2020-02-28 | *null* | *null* | *null* | -3 |
| 2020-03-09 | 2020-03-16 | 2020-03-15 | 2020-03-09 | 4 | 6 | 6 | 7 |
| 2020-03-09 | 2020-03-23 | 2020-03-12 | 2020-03-16 | 8 | 10 | 11 | 14 |
| 2020-03-09 | 2020-04-10 | 2020-04-02 | 2020-04-03 | 20 | 24 | 25 | 32 |
| 2020-03-09 | 2021-03-10 | 2021-03-02 | 2021-03-03 | 210 | 262 | 263 | 366 |

# NETWORKDAYSINTL Function

**Contents:**

Calculates the number of working days between two specified dates. Optionally, you can specify which days of the week are working days as an input parameter. Optional list of holidays can be specified.

- Inputs can be column references or the outputs of the DATE or TIME functions.
    - See *DATE Function*.
    - See *TIME Function*.
- The first value is used as the baseline to compare the date values.
- If the first date value occurs after the second date value, a null value is returned.

**Wrangle vs. SQL:** This function is part of Wrangle , a proprietary data transformation language. Wrangle is not SQL. For more information, see *Wrangle Language*.

## Basic Usage

```
networkdaysintl(StartDate, EndDate)
```

**Output:** Returns the number of working days between `StartDate` and `EndDate`.

## Syntax and Arguments

```
networkdaysintl(date1,date2<span>[</span><span>,</span><span>str_workingdays]</span>
[<span>,</span>array_holiday])
```

| Argument | Required? | Data Type | Description |
|---|---|---|---|
| date1 | Y | datetime | Starting date to compare |
| date2 | Y | datetime | Ending date to compare |
| str_workingdays | N | string | Seven-character string identifying the weekend days. |
| array_holiday | N | array | An array of string values representing the valid dates of holidays. |

For more information on syntax standards, see *Language Documentation Syntax Notes*.

**date1, date2**

Date values can be column references or output of the DATE function or the TIME function.

- For more information, see *DATE Function*.
- For more information, see *TIME Function*.

Date values to compared in working days.

- If `date2` > `date1`, then results are positive.
- If `date2` < `date1`, then a null value is returned.

If `date1` and `date2` have a specified time zone offset, the function calculates the difference including the timezone offsets.

- If `date1` does not have a specified time zone but `date2` does, the function uses the local time in the same time zone as `date2` to calculate the difference. The functions returns the difference without the time zone offset.

**Usage Notes:**

| Required? | Data Type | Example Value |
|---|---|---|
| Yes | Datetime (Column reference or date output of DATE or TIME function) | `LastContactDate` |

### str_workingdays

A seven-character string identifying the days of the week that are working days.

- String value must be seven characters long and contain only `0` or `1` characters. All other values are ignored.
- First character in the string represents Monday and last character in the string represents Sunday.
- If the string is not specified, then a Monday - Friday workweek is used.

Examples:

| str_workingdays | Weekend days |
|---|---|
| `'0000011'` | Saturday and Sunday (default) |
| `'1000011'` | Monday, Saturday, and Sunday |
| `'0000000'` | None. |

**Usage Notes:**

| Required? | Data Type | Example Value |
|---|---|---|
| Yes | Array | ['1000011'] |

### array_holiday

An array containing the list of holidays, which are factored in the calculation of working days.

Values in the array must be in either of the following formats:

```
['2020-12-24','2020-12-25']
['2020/12/24','2020/12/25']
```

**Usage Notes:**

| Required? | Data Type | Example Value |
|---|---|---|
| Yes | Array | ['2018-12-24','2018-12-25','2018-12-31','2019-01-01'] |

## Examples

> **Tip:** For additional examples, see *Common Tasks*.

### Example - Date diffing functions

This example demonstrates how to calculate the number of days between two input dates.

**Functions**:

| Item | Description |
|---|---|
| DATEDIF Function | Calculates the difference between two valid date values for the specified units of measure. |
| NETWORKDAYS Function | Calculates the number of working days between two specified dates, assuming Monday - Friday workweek. Optional list of holidays can be specified. |
| NETWORKDAYS INTL Function | Calculates the number of working days between two specified dates. Optionally, you can specify which days of the week are working days as an input parameter. Optional list of holidays can be specified. |
| WORKDAY Function | Calculates the work date that is before or after a start date, as specified by a number of days. A set of holiday dates can be optionally specified. |
| WORKDAYINTL Function | Calculates the work date that is before or after a start date, as specified by a number of days. You can also specify which days of the week are working days and a list of holidays via parameters. |

**Source:**

The following dataset contains two columns of dates.

- The first column values are constant. This date falls on a Monday.

| Date1 | Date2 |
|---|---|
| 2020-03-09 | 2020-03-13 |
| 2020-03-09 | 2020-03-06 |
| 2020-03-09 | 2020-03-16 |
| 2020-03-09 | 2020-03-23 |
| 2020-03-09 | 2020-04-10 |
| 2020-03-09 | 2021-03-10 |

**Transformation:**

The first transformation calculates the number of raw days between the two dates:

| | |
|---|---|
| **Transformation Name** | `New formula` |
| **Parameter: Formula type** | `Single row formula` |
| **Parameter: Formula** | `datedif(Date1, Date2, day)` |

| Parameter: New column name | `'datedif'` |
|---|---|

This step computes the number of working days between the two dates. Assumptions:

- Workweek is Monday - Friday.
- There are no holidays.

| Transformation Name | `New formula` |
|---|---|
| Parameter: Formula type | `Single row formula` |
| Parameter: Formula | `networkdays(Date1, Date2, [])` |
| Parameter: New column name | `'networkDays'` |

For some, March 17 is an important date, especially if you are Irish. To add St. Patrick's Day to the list of holidays, you could add the following transformation:

| Transformation Name | `New formula` |
|---|---|
| Parameter: Formula type | `Single row formula` |
| Parameter: Formula | `networkdays(Date1, Date2, ['2020-03-17'])` |
| Parameter: New column name | `'networkDaysStPatricks'` |

In the following transformation, the NETWORKDAYSINTL function is applied so that you can specify the working days in the week:

| Transformation Name | `New formula` |
|---|---|
| Parameter: Formula type | `Single row formula` |
| Parameter: Formula | `networkdaysintl(Date1, Date2, '1000011', [])` |
| Parameter: New column name | `'networkDaysIntl'` |

The following two functions enable you to calculate a specific working date based on an input date and integer number of days before or after it. In the following, the date that is five working days before the `Date2` column is computed:

| Transformation Name | `New formula` |
|---|---|
| Parameter: Formula type | `Single row formula` |
| Parameter: Formula | `workday(Date2, -5)` |
| Parameter: New column name | `'workday'` |

Suppose you wish to factor in a four-day workweek, in which Friday through Sunday is considered the weekend:

| Transformation Name | `New formula` |
|---|---|
| Parameter: Formula type | `Single row formula` |

| Parameter: Formula | workdayintl(Date2, -5,'0000111') |
|---|---|
| Parameter: New column name | 'workdayintl' |

**Results:**

| Date1 | Date2 | workdayintl | workday | networkDaysIntl | networkDaysStPatricks | networkDays | datedif |
|---|---|---|---|---|---|---|---|
| 2020-03-09 | 2020-03-13 | 2020-03-05 | 2020-03-06 | 4 | 5 | 5 | 4 |
| 2020-03-09 | 2020-03-06 | 2020-02-27 | 2020-02-28 | *null* | *null* | *null* | -3 |
| 2020-03-09 | 2020-03-16 | 2020-03-15 | 2020-03-09 | 4 | 6 | 6 | 7 |
| 2020-03-09 | 2020-03-23 | 2020-03-12 | 2020-03-16 | 8 | 10 | 11 | 14 |
| 2020-03-09 | 2020-04-10 | 2020-04-02 | 2020-04-03 | 20 | 24 | 25 | 32 |
| 2020-03-09 | 2021-03-10 | 2021-03-02 | 2021-03-03 | 210 | 262 | 263 | 366 |

# MINDATE Function

Computes the minimum value found in all row values in a Datetime column.

If a row contains a missing or null value, it is not factored into the calculation. If no Datetime values are found in the source column, the function returns a null value.

**Wrangle vs. SQL:** This function is part of Wrangle , a proprietary data transformation language. Wrangle is not SQL. For more information, see *Wrangle Language*.

## Basic Usage

```
mindate(myDates)
```

**Output:** Returns the minimum Datetime value from the `myDates` column.

## Syntax and Arguments

```
mindate(function_col_ref)
```

| Argument | Required? | Data Type | Description |
|---|---|---|---|
| function_col_ref | Y | string | Name of column to which to apply the function |

For more information on syntax standards, see *Language Documentation Syntax Notes*.

**function_col_ref**

Name of the column the Datetime values of which you want to calculate the minimum date.

- Column must contain Datetime values.
- Literal values are not supported as inputs.
- Multiple columns and wildcards are not supported.

**Usage Notes:**

| Required? | Data Type | Example Value |
|---|---|---|
| Yes | Datetime (column reference) | `datTransactions` |

## Examples

> **Tip:** For additional examples, see *Common Tasks*.

This example shows how you can apply statistical functions on Datetime columns.

**Functions**:

| Item | Description |
|---|---|
| MINDATE Function | Computes the minimum value found in all row values in a Datetime column. |
| MAXDATE Function | Computes the maximum value found in all row values in a Datetime column. |
| MODEDATE Function | Computes the most frequent (mode) value found in all row values in a Datetime column. |

**Source:**

The following dataset contains a set of three available dates for a set of classes:

| classId | Date1 | Date2 | Date3 |
|---|---|---|---|
| c001 | 2020-03-09 | 2020-03-13 | 2020-03-17 |
| c002 | 2020-03-09 | 2020-03-06 | 2020-03-21 |
| c003 | 2020-03-09 | 2020-03-16 | 2020-03-23 |
| c004 | 2020-03-09 | 2020-03-23 | 2020-04-06 |
| c005 | 2020-03-09 | 2020-04-09 | 2020-05-09 |
| c006 | 2020-03-09 | 2020-08-09 | 2021-01-09 |

**Transformation:**

To compare dates across multiple columns, you must consolidate the values into a single column. You can use the following transformation to do so:

| Transformation Name | Unpivot columns |
|---|---|
| Parameter: Columns | Date1,Date2,Date3 |
| Parameter: Group size | 1 |

The dataset is now contained in three columns, with descriptions listed below:

| classId | key | value |
|---|---|---|
| Same as previous. | DateX column identifier | Corresponding value from the DateX column. |

You can use the following to rename the `value` column to `eventDates`:

| Transformation Name | Rename columns |
|---|---|
| Parameter: Option | Manual rename |
| Parameter: Column | value |
| Parameter: New column name | eventDates |

Using the following transformations, you can create new columns containing the min, max, and mode values for the Datetime values in `eventDates`:

| Transformation Name | New formula |
|---|---|

| Parameter: Formula type | Single row formula |
|---|---|
| Parameter: Formula | MINDATE(eventDates) |
| Parameter: New column name | earliestDate |

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | MAXDATE(eventDates) |
| Parameter: New column name | latestDate |

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | MODEDATE(eventDates) |
| Parameter: New column name | mostFrequentDate |

**Results:**

| classId | key | eventDates | mostFrequentDate | latestDate | earliestDate |
|---|---|---|---|---|---|
| c001 | Date1 | 2020-03-09 | 2020-03-09 | 2021-01-09 | 2020-03-06 |
| c001 | Date2 | 2020-03-13 | 2020-03-09 | 2021-01-09 | 2020-03-06 |
| c001 | Date3 | 2020-03-17 | 2020-03-09 | 2021-01-09 | 2020-03-06 |
| c002 | Date1 | 2020-03-09 | 2020-03-09 | 2021-01-09 | 2020-03-06 |
| c002 | Date2 | 2020-03-06 | 2020-03-09 | 2021-01-09 | 2020-03-06 |
| c002 | Date3 | 2020-03-21 | 2020-03-09 | 2021-01-09 | 2020-03-06 |
| c003 | Date1 | 2020-03-09 | 2020-03-09 | 2021-01-09 | 2020-03-06 |
| c003 | Date2 | 2020-03-16 | 2020-03-09 | 2021-01-09 | 2020-03-06 |
| c003 | Date3 | 2020-03-23 | 2020-03-09 | 2021-01-09 | 2020-03-06 |
| c004 | Date1 | 2020-03-09 | 2020-03-09 | 2021-01-09 | 2020-03-06 |
| c004 | Date2 | 2020-03-23 | 2020-03-09 | 2021-01-09 | 2020-03-06 |
| c004 | Date3 | 2020-04-06 | 2020-03-09 | 2021-01-09 | 2020-03-06 |
| c005 | Date1 | 2020-03-09 | 2020-03-09 | 2021-01-09 | 2020-03-06 |
| c005 | Date2 | 2020-04-09 | 2020-03-09 | 2021-01-09 | 2020-03-06 |
| c005 | Date3 | 2020-05-09 | 2020-03-09 | 2021-01-09 | 2020-03-06 |
| c006 | Date1 | 2020-03-09 | 2020-03-09 | 2021-01-09 | 2020-03-06 |
| c006 | Date2 | 2020-08-09 | 2020-03-09 | 2021-01-09 | 2020-03-06 |
| c006 | Date3 | 2021-01-09 | 2020-03-09 | 2021-01-09 | 2020-03-06 |

# MAXDATE Function

Computes the maximum value found in all row values in a Datetime column.

If a row contains a missing or null value, it is not factored into the calculation. If no Datetime values are found in the source column, the function returns a null value.

**Wrangle vs. SQL:** This function is part of Wrangle , a proprietary data transformation language. Wrangle is not SQL. For more information, see *Wrangle Language*.

## Basic Usage

```
maxdate(myDates)
```

**Output:** Returns the maximum Datetime value from the `myDates` column.

## Syntax and Arguments

```
maxdate(function_col_ref)
```

| Argument | Required? | Data Type | Description |
|---|---|---|---|
| function_col_ref | Y | string | Name of column to which to apply the function |

For more information on syntax standards, see *Language Documentation Syntax Notes*.

**function_col_ref**

Name of the column the Datetime values of which you want to calculate the maximum date.

- Column must contain Datetime values.
- Literal values are not supported as inputs.
- Multiple columns and wildcards are not supported.

**Usage Notes:**

| Required? | Data Type | Example Value |
|---|---|---|
| Yes | Datetime (column reference) | `datTransactions` |

## Examples

> **Tip:** For additional examples, see *Common Tasks*.

This example shows how you can apply statistical functions on Datetime columns.

**Functions**:

| Item | Description |
|---|---|
| MINDATE Function | Computes the minimum value found in all row values in a Datetime column. |
| MAXDATE Function | Computes the maximum value found in all row values in a Datetime column. |
| MODEDATE Function | Computes the most frequent (mode) value found in all row values in a Datetime column. |

**Source:**

The following dataset contains a set of three available dates for a set of classes:

| classId | Date1 | Date2 | Date3 |
|---|---|---|---|
| c001 | 2020-03-09 | 2020-03-13 | 2020-03-17 |
| c002 | 2020-03-09 | 2020-03-06 | 2020-03-21 |
| c003 | 2020-03-09 | 2020-03-16 | 2020-03-23 |
| c004 | 2020-03-09 | 2020-03-23 | 2020-04-06 |
| c005 | 2020-03-09 | 2020-04-09 | 2020-05-09 |
| c006 | 2020-03-09 | 2020-08-09 | 2021-01-09 |

**Transformation:**

To compare dates across multiple columns, you must consolidate the values into a single column. You can use the following transformation to do so:

| | |
|---|---|
| **Transformation Name** | `Unpivot columns` |
| **Parameter: Columns** | `Date1,Date2,Date3` |
| **Parameter: Group size** | `1` |

The dataset is now contained in three columns, with descriptions listed below:

| classId | key | value |
|---|---|---|
| Same as previous. | DateX column identifier | Corresponding value from the DateX column. |

You can use the following to rename the `value` column to `eventDates`:

| | |
|---|---|
| **Transformation Name** | `Rename columns` |
| **Parameter: Option** | `Manual rename` |
| **Parameter: Column** | `value` |
| **Parameter: New column name** | `eventDates` |

Using the following transformations, you can create new columns containing the min, max, and mode values for the Datetime values in `eventDates`:

| | |
|---|---|
| **Transformation Name** | `New formula` |

| Parameter: Formula type | Single row formula |
|---|---|
| Parameter: Formula | MINDATE(eventDates) |
| Parameter: New column name | earliestDate |

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | MAXDATE(eventDates) |
| Parameter: New column name | latestDate |

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | MODEDATE(eventDates) |
| Parameter: New column name | mostFrequentDate |

**Results:**

| classId | key | eventDates | mostFrequentDate | latestDate | earliestDate |
|---|---|---|---|---|---|
| c001 | Date1 | 2020-03-09 | 2020-03-09 | 2021-01-09 | 2020-03-06 |
| c001 | Date2 | 2020-03-13 | 2020-03-09 | 2021-01-09 | 2020-03-06 |
| c001 | Date3 | 2020-03-17 | 2020-03-09 | 2021-01-09 | 2020-03-06 |
| c002 | Date1 | 2020-03-09 | 2020-03-09 | 2021-01-09 | 2020-03-06 |
| c002 | Date2 | 2020-03-06 | 2020-03-09 | 2021-01-09 | 2020-03-06 |
| c002 | Date3 | 2020-03-21 | 2020-03-09 | 2021-01-09 | 2020-03-06 |
| c003 | Date1 | 2020-03-09 | 2020-03-09 | 2021-01-09 | 2020-03-06 |
| c003 | Date2 | 2020-03-16 | 2020-03-09 | 2021-01-09 | 2020-03-06 |
| c003 | Date3 | 2020-03-23 | 2020-03-09 | 2021-01-09 | 2020-03-06 |
| c004 | Date1 | 2020-03-09 | 2020-03-09 | 2021-01-09 | 2020-03-06 |
| c004 | Date2 | 2020-03-23 | 2020-03-09 | 2021-01-09 | 2020-03-06 |
| c004 | Date3 | 2020-04-06 | 2020-03-09 | 2021-01-09 | 2020-03-06 |
| c005 | Date1 | 2020-03-09 | 2020-03-09 | 2021-01-09 | 2020-03-06 |
| c005 | Date2 | 2020-04-09 | 2020-03-09 | 2021-01-09 | 2020-03-06 |
| c005 | Date3 | 2020-05-09 | 2020-03-09 | 2021-01-09 | 2020-03-06 |
| c006 | Date1 | 2020-03-09 | 2020-03-09 | 2021-01-09 | 2020-03-06 |
| c006 | Date2 | 2020-08-09 | 2020-03-09 | 2021-01-09 | 2020-03-06 |
| c006 | Date3 | 2021-01-09 | 2020-03-09 | 2021-01-09 | 2020-03-06 |

# MODEDATE Function

Computes the most frequent (mode) value found in all row values in a Datetime column.

If a row contains a missing or null value, it is not factored into the calculation. If no Datetime values are found in the source column, the function returns a null value.

**Wrangle vs. SQL:** This function is part of Wrangle , a proprietary data transformation language. Wrangle is not SQL. For more information, see *Wrangle Language*.

## Basic Usage

```
modedate(myDates)
```

**Output:** Returns the most frequently appearing Datetime value from the `myDates` column.

## Syntax and Arguments

```
modedate(function_col_ref)
```

| Argument | Required? | Data Type | Description |
|---|---|---|---|
| function_col_ref | Y | string | Name of column to which to apply the function |

For more information on syntax standards, see *Language Documentation Syntax Notes*.

**function_col_ref**

Name of the column the Datetime values of which you want to calculate the most frequent (mode) date.

- Column must contain Datetime values.
- Literal values are not supported as inputs.
- Multiple columns and wildcards are not supported.

**Usage Notes:**

| Required? | Data Type | Example Value |
|---|---|---|
| Yes | Datetime (column reference) | `datTransactions` |

## Examples

> **Tip:** For additional examples, see *Common Tasks*.

This example shows how you can apply statistical functions on Datetime columns.

**Functions**:

| Item | Description |
|------|-------------|
| MINDATE Function | Computes the minimum value found in all row values in a Datetime column. |
| MAXDATE Function | Computes the maximum value found in all row values in a Datetime column. |
| MODEDATE Function | Computes the most frequent (mode) value found in all row values in a Datetime column. |

**Source:**

The following dataset contains a set of three available dates for a set of classes:

| classId | Date1 | Date2 | Date3 |
|---------|-------|-------|-------|
| c001 | 2020-03-09 | 2020-03-13 | 2020-03-17 |
| c002 | 2020-03-09 | 2020-03-06 | 2020-03-21 |
| c003 | 2020-03-09 | 2020-03-16 | 2020-03-23 |
| c004 | 2020-03-09 | 2020-03-23 | 2020-04-06 |
| c005 | 2020-03-09 | 2020-04-09 | 2020-05-09 |
| c006 | 2020-03-09 | 2020-08-09 | 2021-01-09 |

**Transformation:**

To compare dates across multiple columns, you must consolidate the values into a single column. You can use the following transformation to do so:

| | |
|------|------|
| **Transformation Name** | `Unpivot columns` |
| **Parameter: Columns** | `Date1,Date2,Date3` |
| **Parameter: Group size** | `1` |

The dataset is now contained in three columns, with descriptions listed below:

| classId | key | value |
|---------|-----|-------|
| Same as previous. | DateX column identifier | Corresponding value from the DateX column. |

You can use the following to rename the `value` column to `eventDates`:

| | |
|------|------|
| **Transformation Name** | `Rename columns` |
| **Parameter: Option** | `Manual rename` |
| **Parameter: Column** | `value` |
| **Parameter: New column name** | `eventDates` |

Using the following transformations, you can create new columns containing the min, max, and mode values for the Datetime values in `eventDates`:

| | |
|------|------|
| **Transformation Name** | `New formula` |

| Parameter: Formula type | Single row formula |
|---|---|
| Parameter: Formula | MINDATE(eventDates) |
| Parameter: New column name | earliestDate |

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | MAXDATE(eventDates) |
| Parameter: New column name | latestDate |

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | MODEDATE(eventDates) |
| Parameter: New column name | mostFrequentDate |

**Results:**

| classId | key | eventDates | mostFrequentDate | latestDate | earliestDate |
|---|---|---|---|---|---|
| c001 | Date1 | 2020-03-09 | 2020-03-09 | 2021-01-09 | 2020-03-06 |
| c001 | Date2 | 2020-03-13 | 2020-03-09 | 2021-01-09 | 2020-03-06 |
| c001 | Date3 | 2020-03-17 | 2020-03-09 | 2021-01-09 | 2020-03-06 |
| c002 | Date1 | 2020-03-09 | 2020-03-09 | 2021-01-09 | 2020-03-06 |
| c002 | Date2 | 2020-03-06 | 2020-03-09 | 2021-01-09 | 2020-03-06 |
| c002 | Date3 | 2020-03-21 | 2020-03-09 | 2021-01-09 | 2020-03-06 |
| c003 | Date1 | 2020-03-09 | 2020-03-09 | 2021-01-09 | 2020-03-06 |
| c003 | Date2 | 2020-03-16 | 2020-03-09 | 2021-01-09 | 2020-03-06 |
| c003 | Date3 | 2020-03-23 | 2020-03-09 | 2021-01-09 | 2020-03-06 |
| c004 | Date1 | 2020-03-09 | 2020-03-09 | 2021-01-09 | 2020-03-06 |
| c004 | Date2 | 2020-03-23 | 2020-03-09 | 2021-01-09 | 2020-03-06 |
| c004 | Date3 | 2020-04-06 | 2020-03-09 | 2021-01-09 | 2020-03-06 |
| c005 | Date1 | 2020-03-09 | 2020-03-09 | 2021-01-09 | 2020-03-06 |
| c005 | Date2 | 2020-04-09 | 2020-03-09 | 2021-01-09 | 2020-03-06 |
| c005 | Date3 | 2020-05-09 | 2020-03-09 | 2021-01-09 | 2020-03-06 |
| c006 | Date1 | 2020-03-09 | 2020-03-09 | 2021-01-09 | 2020-03-06 |
| c006 | Date2 | 2020-08-09 | 2020-03-09 | 2021-01-09 | 2020-03-06 |
| c006 | Date3 | 2021-01-09 | 2020-03-09 | 2021-01-09 | 2020-03-06 |

# WORKDAY Function

**Contents:**

---

Calculates the work date that is before or after a start date, as specified by a number of days. A set of holiday dates can be optionally specified.

- Input can be a column reference or the output of the DATE or TIME function.
    - See *DATE Function*.
    - See *TIME Function*.
- The first value is used as the baseline.
- The second value is the number of days before or after the start date.
    - If the second value is negative, the function returns the number of days before the start date.

**Wrangle vs. SQL:** This function is part of Wrangle , a proprietary data transformation language. Wrangle is not SQL. For more information, see *Wrangle Language*.

## Basic Usage

```
workday(StartDate, 4)
```

**Output:** Returns the date that is four working days after `StartDate`.

## Syntax and Arguments

```
workday(date1,numDays,[array_holiday])
```

| Argument | Required? | Data Type | Description |
|---|---|---|---|
| date1 | Y | datetime | Starting date to compare |
| numDays | Y | integer | Number of days before or after starting date |
| array_holiday | N | array | An array of string values representing the valid dates of holidays |

For more information on syntax standards, see *Language Documentation Syntax Notes*.

### date1

Date value can be column references or output of the `DATE` function or the `TIME` function.

- For more information, see *DATE Function*.
- For more information, see *TIME Function*.

**Usage Notes:**

| Required? | Data Type | Example Value |
|---|---|---|
| Yes | Datetime (Column reference or date output of DATE or TIME function) | `LastContactDate` |

### numDays

An Integer that defines the number of working days distance from the start date. The function returns the start date plus or minus the number of working days represented in this Integer.

If the integer is less than zero, the number of working days are counted backward from the start date.

**Usage Notes:**

| Required? | Data Type | Example Value |
|---|---|---|
| Yes | integer | 10 |

### array_holiday

An array containing the list of holidays, which are factored in the calculation of working days.

Values in the array must be in either of the following formats:

```
['2020-12-24','2020-12-25']
['2020/12/24','2020/12/25']
```

**Usage Notes:**

| Required? | Data Type | Example Value |
|---|---|---|
| Yes | Array | ['2018-12,24','2018-12-25','2018-12-31','2019-01-01'] |

## Examples

> **Tip:** For additional examples, see *Common Tasks.*

### Example - Date diffing functions

This example demonstrates how to calculate the number of days between two input dates.

**Functions**:

| Item | Description |
|---|---|
| DATEDIF Function | Calculates the difference between two valid date values for the specified units of measure. |
| NETWORKDAYS Function | Calculates the number of working days between two specified dates, assuming Monday - Friday workweek. Optional list of holidays can be specified. |

| NETWORKDAYS INTL Function | Calculates the number of working days between two specified dates. Optionally, you can specify which days of the week are working days as an input parameter. Optional list of holidays can be specified. |
|---|---|
| WORKDAY Function | Calculates the work date that is before or after a start date, as specified by a number of days. A set of holiday dates can be optionally specified. |
| WORKDAYINTL Function | Calculates the work date that is before or after a start date, as specified by a number of days. You can also specify which days of the week are working days and a list of holidays via parameters. |

**Source:**

The following dataset contains two columns of dates.

- The first column values are constant. This date falls on a Monday.

| Date1 | Date2 |
|---|---|
| 2020-03-09 | 2020-03-13 |
| 2020-03-09 | 2020-03-06 |
| 2020-03-09 | 2020-03-16 |
| 2020-03-09 | 2020-03-23 |
| 2020-03-09 | 2020-04-10 |
| 2020-03-09 | 2021-03-10 |

**Transformation:**

The first transformation calculates the number of raw days between the two dates:

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | datedif(Date1, Date2, day) |
| Parameter: New column name | 'datedif' |

This step computes the number of working days between the two dates. Assumptions:

- Workweek is Monday - Friday.
- There are no holidays.

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | networkdays(Date1, Date2, []) |
| Parameter: New column name | 'networkDays' |

For some, March 17 is an important date, especially if you are Irish. To add St. Patrick's Day to the list of holidays, you could add the following transformation:

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |

| Parameter: Formula | `networkdays(Date1, Date2, ['2020-03-17'])` |
| --- | --- |
| Parameter: New column name | `'networkDaysStPatricks'` |

In the following transformation, the NETWORKDAYSINTL function is applied so that you can specify the working days in the week:

| Transformation Name | `New formula` |
| --- | --- |
| Parameter: Formula type | `Single row formula` |
| Parameter: Formula | `networkdaysintl(Date1, Date2, '1000011', [])` |
| Parameter: New column name | `'networkDaysIntl'` |

The following two functions enable you to calculate a specific working date based on an input date and integer number of days before or after it. In the following, the date that is five working days before the `Date2` column is computed:

| Transformation Name | `New formula` |
| --- | --- |
| Parameter: Formula type | `Single row formula` |
| Parameter: Formula | `workday(Date2, -5)` |
| Parameter: New column name | `'workday'` |

Suppose you wish to factor in a four-day workweek, in which Friday through Sunday is considered the weekend:

| Transformation Name | `New formula` |
| --- | --- |
| Parameter: Formula type | `Single row formula` |
| Parameter: Formula | `workdayintl(Date2, -5,'0000111')` |
| Parameter: New column name | `'workdayintl'` |

**Results:**

| Date1 | Date2 | workdayintl | workday | networkDaysIntl | networkDaysStPatricks | networkDays | datedif |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 2020-03-09 | 2020-03-13 | 2020-03-05 | 2020-03-06 | 4 | 5 | 5 | 4 |
| 2020-03-09 | 2020-03-06 | 2020-02-27 | 2020-02-28 | *null* | *null* | *null* | -3 |
| 2020-03-09 | 2020-03-16 | 2020-03-15 | 2020-03-09 | 4 | 6 | 6 | 7 |
| 2020-03-09 | 2020-03-23 | 2020-03-12 | 2020-03-16 | 8 | 10 | 11 | 14 |
| 2020-03-09 | 2020-04-10 | 2020-04-02 | 2020-04-03 | 20 | 24 | 25 | 32 |
| 2020-03-09 | 2021-03-10 | 2021-03-02 | 2021-03-03 | 210 | 262 | 263 | 366 |

# WORKDAYINTL Function

**Contents:**

---

Calculates the work date that is before or after a start date, as specified by a number of days. You can also specify which days of the week are working days and a list of holidays via parameters.

- Input can be a column reference or the output of the DATE or TIME function.
    - See *DATE Function*.
    - See *TIME Function*.
- The first value is used as the baseline.
- The second value is the number of days before or after the start date.
    - If the second value is negative, the function returns the number of days before the start date.

**Wrangle vs. SQL:** This function is part of Wrangle , a proprietary data transformation language. Wrangle is not SQL. For more information, see *Wrangle Language*.

## Basic Usage

```
workdayintl(StartDate, 5,&apos;0000001&apos;)
```

**Output:** Returns the working date that is five days after the `StartDate`, assuming that every day except for Sunday is a working day.

## Syntax and Arguments

```
workdayintl(date1,numDays<span>[</span><span>,</span><span>str_workingdays]</span>[<span>,<
/span>array_holiday])
```

| Argument | Required? | Data Type | Description |
|---|---|---|---|
| date1 | Y | datetime | Starting date to compare |
| numDays | Y | integer | Number of days before or after starting date |
| str_workingdays | N | string | Seven-character string identifying the weekend days. |
| array_holiday | N | array | An array of string values representing the valid dates of holidays. |

For more information on syntax standards, see *Language Documentation Syntax Notes*.

### date1

Date value can be a column reference or output of the `DATE` function or the `TIME` function.

- For more information, see *DATE Function*.
- For more information, see *TIME Function*.

**Usage Notes:**

| Required? | Data Type | Example Value |
|---|---|---|
| Yes | Datetime (Column reference or date output of DATE or TIME function) | `LastContactDate` |

### numDays

An Integer that defines the number of working days distance from the start date. The function returns the start date plus or minus the number of working days represented in this Integer.

If the integer is less than zero, the number of working days are counted backward from the start date.

**Usage Notes:**

| Required? | Data Type | Example Value |
|---|---|---|
| Yes | integer | 10 |

### str_workingdays

A seven-character string identifying the days of the week that are working days.

- String value must be seven characters long and contain only `0` or `1` characters. All other values are ignored.
- First character in the string represents Monday and last character in the string represents Sunday.
- If the string is not specified, then a Monday - Friday workweek is used.

Examples:

| str_workingdays | Weekend days |
|---|---|
| `'0000011'` | Saturday and Sunday (default) |
| `'1000011'` | Monday, Saturday, and Sunday |
| `'0000000'` | None. |

**Usage Notes:**

| Required? | Data Type | Example Value |
|---|---|---|
| Yes | Array | ['1000011'] |

### array_holiday

An array containing the list of holidays, which are factored in the calculation of working days.

Values in the array must be in either of the following formats:

```
['2020-12-24','2020-12-25']
['2020/12/24','2020/12/25']
```

**Usage Notes:**

| Required? | Data Type | Example Value |
|-----------|-----------|---------------|
| Yes | Array | ['2018-12-24','2018-12-25','2018-12-31','2019-01-01'] |

## Examples

> **Tip:** For additional examples, see *Common Tasks*.

### Example - Date diffing functions

This example demonstrates how to calculate the number of days between two input dates.

**Functions**:

| Item | Description |
|------|-------------|
| DATEDIF Function | Calculates the difference between two valid date values for the specified units of measure. |
| NETWORKDAYS Function | Calculates the number of working days between two specified dates, assuming Monday - Friday workweek. Optional list of holidays can be specified. |
| NETWORKDAYS INTL Function | Calculates the number of working days between two specified dates. Optionally, you can specify which days of the week are working days as an input parameter. Optional list of holidays can be specified. |
| WORKDAY Function | Calculates the work date that is before or after a start date, as specified by a number of days. A set of holiday dates can be optionally specified. |
| WORKDAYINTL Function | Calculates the work date that is before or after a start date, as specified by a number of days. You can also specify which days of the week are working days and a list of holidays via parameters. |

**Source:**

The following dataset contains two columns of dates.

- The first column values are constant. This date falls on a Monday.

| Date1 | Date2 |
|-------|-------|
| 2020-03-09 | 2020-03-13 |
| 2020-03-09 | 2020-03-06 |
| 2020-03-09 | 2020-03-16 |
| 2020-03-09 | 2020-03-23 |
| 2020-03-09 | 2020-04-10 |
| 2020-03-09 | 2021-03-10 |

**Transformation:**

The first transformation calculates the number of raw days between the two dates:

| Transformation Name | New formula |
|---------------------|-------------|
| **Parameter: Formula type** | Single row formula |

| Parameter: Formula | datedif(Date1, Date2, day) |
|---|---|
| Parameter: New column name | 'datedif' |

This step computes the number of working days between the two dates. Assumptions:

- Workweek is Monday - Friday.
- There are no holidays.

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | networkdays(Date1, Date2, []) |
| Parameter: New column name | 'networkDays' |

For some, March 17 is an important date, especially if you are Irish. To add St. Patrick's Day to the list of holidays, you could add the following transformation:

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | networkdays(Date1, Date2, ['2020-03-17']) |
| Parameter: New column name | 'networkDaysStPatricks' |

In the following transformation, the NETWORKDAYSINTL function is applied so that you can specify the working days in the week:

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | networkdaysintl(Date1, Date2, '1000011', []) |
| Parameter: New column name | 'networkDaysIntl' |

The following two functions enable you to calculate a specific working date based on an input date and integer number of days before or after it. In the following, the date that is five working days before the `Date2` column is computed:

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | workday(Date2, -5) |
| Parameter: New column name | 'workday' |

Suppose you wish to factor in a four-day workweek, in which Friday through Sunday is considered the weekend:

| Transformation Name | New formula |
|---|---|

| Parameter: Formula type | Single row formula |
|---|---|
| Parameter: Formula | workdayintl(Date2, -5,'0000111') |
| Parameter: New column name | 'workdayintl' |

**Results:**

| Date1 | Date2 | workdayintl | workday | networkDaysIntl | networkDaysStPatricks | networkDays | datedif |
|---|---|---|---|---|---|---|---|
| 2020-03-09 | 2020-03-13 | 2020-03-05 | 2020-03-06 | 4 | 5 | 5 | 4 |
| 2020-03-09 | 2020-03-06 | 2020-02-27 | 2020-02-28 | *null* | *null* | *null* | -3 |
| 2020-03-09 | 2020-03-16 | 2020-03-15 | 2020-03-09 | 4 | 6 | 6 | 7 |
| 2020-03-09 | 2020-03-23 | 2020-03-12 | 2020-03-16 | 8 | 10 | 11 | 14 |
| 2020-03-09 | 2020-04-10 | 2020-04-02 | 2020-04-03 | 20 | 24 | 25 | 32 |
| 2020-03-09 | 2021-03-10 | 2021-03-02 | 2021-03-03 | 210 | 262 | 263 | 366 |

# CONVERTFROMUTC Function

**Contents:**

Converts Datetime value to corresponding value of the specified time zone. Input can be a column of Datetime values, a literal Datetime value, or a function returning Datetime values.

- input Datetime value is assumed to be in UTC time zone. Inputs with time zone offsets are invalid.
- Specified time zone must be a string literal of one of the supported time zone values.

**Wrangle vs. SQL:** This function is part of Wrangle , a proprietary data transformation language. Wrangle is not SQL. For more information, see *Wrangle Language*.

## Basic Usage

**Column reference values:**

```
convertfromutc(myUTCtimestamp,&apos;US/Eastern&apos;)
```

**Output:** Returns the values of the `myUTCtimestamp` converted to US Eastern time zone.

## Syntax and Arguments

```
<span>convertfromutc</span><span>(date, &apos;enum-timezone&apos;)</span>
```

| Argument | Required? | Data Type | Description |
|----------|-----------|-----------|-------------|
| date | Y | datetime | Name of Datetime column, Datetime literal, or function returning a Datetime value. |
| enum-timezone-string | Y | string | Case-sensitive string literal value corresponding to the target time zone. |

For more information on syntax standards, see *Language Documentation Syntax Notes*.

**date**

Name of a column containing Datetime values, a literal Datetime value, or a function returning Datetime values to convert.

> **Tip:** Use the DATEFORMAT function to wrap values into acceptable formats. See
> *DATEFORMAT Function.*

Values are assumed to be in UTC time zone format. **Coordinated Universal Time** is the primary standard time by which clocks are coordinated around the world.

- UTC is also known as Greenwich Mean Time.
- UTC does not change for daylight savings time.
- For more information, see *https://en.wikipedia.org/wiki/Coordinated_Universal_Time*.

If an input value is invalid for Datetime data type, a null value is returned.

- Column references with time zone offsets are invalid.
- Missing values for this function in the source data result in missing values in the output.
- Multiple columns and wildcards are not supported.

**Usage Notes:**

| Required? | Data Type | Example Value |
|---|---|---|
| Yes | Datetime (column reference, function, or literal) | `sourceTime` |

**enum-timezone-string**

String literal value for the time zone to which to convert.

> **NOTE:** These values are case-sensitive.

Example values:

```
'America/Puerto_Rico'
'US/Eastern'
'US/Central'
'US/Mountain'
'US/Pacific'
'US/Alaska'
'US/Hawaii'
```

## Examples

> **Tip:** For additional examples, see *Common Tasks*.

### Example - Time zone conversion

This example shows how you can use functions to convert Datetime values to different time zones.

**Functions**:

| Item | Description |
|---|---|
| CONVERTFROM UTC Function | Converts Datetime value to corresponding value of the specified time zone. Input can be a column of Datetime values, a literal Datetime value, or a function returning Datetime values. |
| CONVERTTOUT C Function | Converts Datetime value in specified time zone to corresponding value in UTC time zone. Input can be a column of Datetime values, a literal Datetime value, or a function returning Datetime values. |
| CONVERTTIMEZ ONE Function | Converts Datetime value in specified time zone to corresponding value second specified time zone. Input can be a column of Datetime values, a literal Datetime value, or a function returning Datetime values. |

| ISMISMATCHED Function | Tests whether a set of values is not valid for a specified data type. |
|---|---|

**Source:**

| row | datetime |
|---|---|
| 1 | 2020-03-15 |
| 2 | 2020-03-15 0:00:00 |
| 3 | 2020-03-15 +08:00 |
| 4 | 2020-03-15 1:02:03 |
| 5 | 2020-03-15 4:02:03 |
| 6 | 2020-03-15 8:02:03 |
| 7 | 2020-03-15 12:02:03 |
| 8 | 2020-03-15 16:02:03 |
| 9 | 2020-03-15 20:02:03 |
| 10 | 2020-03-15 23:02:03 |

**Transformation:**

When you import the above dates, Designer Cloud Powered by Trifacta Enterprise Edition may not recognize the column as a set of dates. You can use the column menus to format the date values to the following standardized format:

```
yyyy*mm*dd*HH:MM:SS
```

| Transformation Name | Change column data type |
|---|---|
| Parameter: Columns | datetime |
| Parameter: New type | Date/Time |
| Parameter: Date/Time type | yyyy*mm*dd*HH:MM:SS |

When the type has been changed, row 1 and row 3 have been identified as invalid. You can use the following transformation to remove these rows:

| Transformation Name | Filter rows |
|---|---|
| Parameter: Condition | Custom formula |
| Parameter: Type of formula | Custom single |
| Parameter: Condition | ISMISMATCHED(datetime, ['Datetime','yy-mm-dd hh:mm:ss','yyyy*mm*dd*HH:MM:SS']) |
| Parameter: Action | Delete matching rows |

When the Datetime values are consistently formatted, you can use the following transformations to perform conversions. The following tranformation converts the values from UTC to US/Eastern time zone:

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | CONVERTFROMUTC(datetime, 'US\/Eastern') |
| Parameter: New column name | 'datetimeUTC2Eastern' |

This transformation now assumes that the date values are in US/Pacific time zone and converts them to UTC:

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | CONVERTTOUTC(datetime, 'US\/Pacific') |
| Parameter: New column name | 'datetimePacific2UTC' |

The final transformation converts the date time values between arbitrary time zones. In this case, the values are assumed to be in US/Alaska time zone and are converted to US/Hawaii time zone:

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | CONVERTTIMEZONE(datetime, 'US\/Alaska', 'US\/Hawaii') |
| Parameter: New column name | 'datetimeAlaska2Hawaii' |

**Results:**

| row | datetime | datetimeAlaska2Hawaii | datetimePacific2UTC | datetimeUTC2Eastern |
|---|---|---|---|---|
| 2 | 2020-03-15 00:00:00 | 2020-03-14 22:00:00 | 2020-03-15 07:00:00 | 2020-03-14 20:00:00 |
| 4 | 2020-03-15 01:02:03 | 2020-03-14 23:02:03 | 2020-03-15 08:02:03 | 2020-03-14 21:02:03 |
| 5 | 2020-03-15 04:02:03 | 2020-03-15 02:02:03 | 2020-03-15 11:02:03 | 2020-03-15 00:02:03 |
| 6 | 2020-03-15 08:02:03 | 2020-03-15 06:02:03 | 2020-03-15 15:02:03 | 2020-03-15 04:02:03 |
| 7 | 2020-03-15 12:02:03 | 2020-03-15 10:02:03 | 2020-03-15 19:02:03 | 2020-03-15 08:02:03 |
| 8 | 2020-03-15 16:02:03 | 2020-03-15 14:02:03 | 2020-03-15 23:02:03 | 2020-03-15 12:02:03 |
| 9 | 2020-03-15 20:02:03 | 2020-03-15 18:02:03 | 2020-03-16 03:02:03 | 2020-03-15 16:02:03 |
| 10 | 2020-03-15 23:02:03 | 2020-03-15 21:02:03 | 2020-03-16 06:02:03 | 2020-03-15 19:02:03 |

# CONVERTTOUTC Function

**Contents:**

Converts Datetime value in specified time zone to corresponding value in UTC time zone. Input can be a column of Datetime values, a literal Datetime value, or a function returning Datetime values.

- Inputs with time zone offsets are invalid.
- Specified time zone must be a string literal of one of the supported time zone values.

**Wrangle vs. SQL:** This function is part of Wrangle , a proprietary data transformation language. Wrangle is not SQL. For more information, see *Wrangle Language*.

## Basic Usage

**Column reference values:**

```
converttoutc(myTimestamp,&apos;US/Mountain&apos;)
```

**Output:** Returns the UTC values of the `myTimestamp` converted from US Mountain time zone.

## Syntax and Arguments

```
<span>converttoutc</span><span>(date, &apos;enum-timezone&apos;)</span>
```

| Argument | Required? | Data Type | Description |
|---|---|---|---|
| date | Y | datetime | Name of Datetime column, Datetime literal, or function returning a Datetime value. |
| enum-timezone-string | Y | string | Case-sensitive string literal value corresponding to the source time zone. |

For more information on syntax standards, see *Language Documentation Syntax Notes*.

**date**

Name of a column containing Datetime values, a literal Datetime value, or a function returning Datetime values to convert.

> **Tip:** Use the DATEFORMAT function to wrap values into acceptable formats. See *DATEFORMAT Function*.

If an input value is invalid for Datetime data type, a null value is returned.

- Column references with time zone offsets are invalid.

- Missing values for this function in the source data result in missing values in the output.
- Multiple columns and wildcards are not supported.

**Usage Notes:**

| Required? | Data Type | Example Value |
|-----------|-----------|---------------|
| Yes | Datetime (column reference, function, or literal) | `sourceTime` |

**enum-timezone-string**

String literal value for the time zone from which to convert.

> **NOTE:** These values are case-sensitive.

Example values:

```
'America/Puerto_Rico'
'US/Eastern'
'US/Central'
'US/Mountain'
'US/Pacific'
'US/Alaska'
'US/Hawaii'
```

## Examples

> **Tip:** For additional examples, see *Common Tasks*.

**Example - Time zone conversion**

This example shows how you can use functions to convert Datetime values to different time zones.

**Functions**:

| Item | Description |
|------|-------------|
| CONVERTFROM UTC Function | Converts Datetime value to corresponding value of the specified time zone. Input can be a column of Datetime values, a literal Datetime value, or a function returning Datetime values. |
| CONVERTTOUT C Function | Converts Datetime value in specified time zone to corresponding value in UTC time zone. Input can be a column of Datetime values, a literal Datetime value, or a function returning Datetime values. |
| CONVERTTIMEZ ONE Function | Converts Datetime value in specified time zone to corresponding value second specified time zone. Input can be a column of Datetime values, a literal Datetime value, or a function returning Datetime values. |
| ISMISMATCHED Function | Tests whether a set of values is not valid for a specified data type. |

**Source:**

| row | datetime |
|-----|----------|
| 1 | 2020-03-15 |

| 2 | 2020-03-15 0:00:00 |
|---|---|
| 3 | 2020-03-15 +08:00 |
| 4 | 2020-03-15 1:02:03 |
| 5 | 2020-03-15 4:02:03 |
| 6 | 2020-03-15 8:02:03 |
| 7 | 2020-03-15 12:02:03 |
| 8 | 2020-03-15 16:02:03 |
| 9 | 2020-03-15 20:02:03 |
| 10 | 2020-03-15 23:02:03 |

**Transformation:**

When you import the above dates, Designer Cloud Powered by Trifacta Enterprise Edition may not recognize the column as a set of dates. You can use the column menus to format the date values to the following standardized format:

```
yyyy*mm*dd*HH:MM:SS
```

| Transformation Name | Change column data type |
|---|---|
| Parameter: Columns | datetime |
| Parameter: New type | Date/Time |
| Parameter: Date/Time type | yyyy*mm*dd*HH:MM:SS |

When the type has been changed, row 1 and row 3 have been identified as invalid. You can use the following transformation to remove these rows:

| Transformation Name | Filter rows |
|---|---|
| Parameter: Condition | Custom formula |
| Parameter: Type of formula | Custom single |
| Parameter: Condition | ISMISMATCHED(datetime, ['Datetime','yy-mm-dd hh:mm:ss','yyyy*mm*dd*HH:MM:SS']) |
| Parameter: Action | Delete matching rows |

When the Datetime values are consistently formatted, you can use the following transformations to perform conversions. The following tranformation converts the values from UTC to US/Eastern time zone:

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | CONVERTFROMUTC(datetime, 'US\/Eastern') |

| Parameter: New column name | 'datetimeUTC2Eastern' |
|---|---|

This transformation now assumes that the date values are in US/Pacific time zone and converts them to UTC:

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | CONVERTTOUTC(datetime, 'US\/Pacific') |
| Parameter: New column name | 'datetimePacific2UTC' |

The final transformation converts the date time values between arbitrary time zones. In this case, the values are assumed to be in US/Alaska time zone and are converted to US/Hawaii time zone:

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | CONVERTTIMEZONE(datetime, 'US\/Alaska', 'US\/Hawaii') |
| Parameter: New column name | 'datetimeAlaska2Hawaii' |

**Results:**

| row | datetime | datetimeAlaska2Hawaii | datetimePacific2UTC | datetimeUTC2Eastern |
|---|---|---|---|---|
| 2 | 2020-03-15 00:00:00 | 2020-03-14 22:00:00 | 2020-03-15 07:00:00 | 2020-03-14 20:00:00 |
| 4 | 2020-03-15 01:02:03 | 2020-03-14 23:02:03 | 2020-03-15 08:02:03 | 2020-03-14 21:02:03 |
| 5 | 2020-03-15 04:02:03 | 2020-03-15 02:02:03 | 2020-03-15 11:02:03 | 2020-03-15 00:02:03 |
| 6 | 2020-03-15 08:02:03 | 2020-03-15 06:02:03 | 2020-03-15 15:02:03 | 2020-03-15 04:02:03 |
| 7 | 2020-03-15 12:02:03 | 2020-03-15 10:02:03 | 2020-03-15 19:02:03 | 2020-03-15 08:02:03 |
| 8 | 2020-03-15 16:02:03 | 2020-03-15 14:02:03 | 2020-03-15 23:02:03 | 2020-03-15 12:02:03 |
| 9 | 2020-03-15 20:02:03 | 2020-03-15 18:02:03 | 2020-03-16 03:02:03 | 2020-03-15 16:02:03 |
| 10 | 2020-03-15 23:02:03 | 2020-03-15 21:02:03 | 2020-03-16 06:02:03 | 2020-03-15 19:02:03 |

# CONVERTTIMEZONE Function

**Contents:**

---

Converts Datetime value in specified time zone to corresponding value second specified time zone. Input can be a column of Datetime values, a literal Datetime value, or a function returning Datetime values.

- Inputs with time zone offsets are invalid.
- Specified time zone must be a string literal of one of the supported time zone values.

**Wrangle vs. SQL:** This function is part of Wrangle , a proprietary data transformation language. Wrangle is not SQL. For more information, see *Wrangle Language*.

## Basic Usage

**Column reference values:**

```
converttimezone(myTimestamp,&apos;US/Mountain&apos;,&apos;US/Pacific&apos;)
```

**Output:** Returns the UTC values of the `myTimestamp` converted from US Mountain time zone to US Pacific time zone.

## Syntax and Arguments

```
<span>converttimezone</span><span>(date, &apos;enum-timezone1&apos;,<span>&apos;enum-timezone2&apos;</span>)</span>
```

| Argument | Required? | Data Type | Description |
|---|---|---|---|
| date | Y | datetime | Name of Datetime column, Datetime literal, or function returning a Datetime value. |
| enum-timezone-string1, enum-timezone-string2 | Y | string | Case-sensitive string literal value corresponding to the source or target time zone. |

For more information on syntax standards, see *Language Documentation Syntax Notes*.

**date**

Name of a column containing Datetime values, a literal Datetime value, or a function returning Datetime values to convert.

> **Tip:** Use the DATEFORMAT function to wrap values into acceptable formats. See *DATEFORMAT Function*.

If an input value is invalid for Datetime data type, a null value is returned.

- Column references with time zone offsets are invalid.
- Missing values for this function in the source data result in missing values in the output.
- Multiple columns and wildcards are not supported.

**Usage Notes:**

| Required? | Data Type | Example Value |
|---|---|---|
| Yes | Datetime (column reference, function, or literal) | `sourceTime` |

### enum-timezone-string1, enum-timezone-string2

String literal value for the time zone 1) to convert from and 2) to convert to.

> **NOTE:** These values are case-sensitive.

Example values:

```
'America/Puerto_Rico'
'US/Eastern'
'US/Central'
'US/Mountain'
'US/Pacific'
'US/Alaska'
'US/Hawaii'
```

## Examples

> **Tip:** For additional examples, see *Common Tasks*.

### Example - Time zone conversion

This example shows how you can use functions to convert Datetime values to different time zones.

**Functions**:

| Item | Description |
|---|---|
| CONVERTFROM UTC Function | Converts Datetime value to corresponding value of the specified time zone. Input can be a column of Datetime values, a literal Datetime value, or a function returning Datetime values. |
| CONVERTTOUT C Function | Converts Datetime value in specified time zone to corresponding value in UTC time zone. Input can be a column of Datetime values, a literal Datetime value, or a function returning Datetime values. |
| CONVERTTIMEZ ONE Function | Converts Datetime value in specified time zone to corresponding value second specified time zone. Input can be a column of Datetime values, a literal Datetime value, or a function returning Datetime values. |
| ISMISMATCHED Function | Tests whether a set of values is not valid for a specified data type. |

**Source:**

| row | datetime |
|-----|----------|
| 1 | 2020-03-15 |
| 2 | 2020-03-15 0:00:00 |
| 3 | 2020-03-15 +08:00 |
| 4 | 2020-03-15 1:02:03 |
| 5 | 2020-03-15 4:02:03 |
| 6 | 2020-03-15 8:02:03 |
| 7 | 2020-03-15 12:02:03 |
| 8 | 2020-03-15 16:02:03 |
| 9 | 2020-03-15 20:02:03 |
| 10 | 2020-03-15 23:02:03 |

**Transformation:**

When you import the above dates, Designer Cloud Powered by Trifacta Enterprise Edition may not recognize the column as a set of dates. You can use the column menus to format the date values to the following standardized format:

```
yyyy*mm*dd*HH:MM:SS
```

| Transformation Name | Change column data type |
|---------------------|-------------------------|
| **Parameter: Columns** | datetime |
| **Parameter: New type** | Date/Time |
| **Parameter: Date/Time type** | yyyy*mm*dd*HH:MM:SS |

When the type has been changed, row 1 and row 3 have been identified as invalid. You can use the following transformation to remove these rows:

| Transformation Name | Filter rows |
|---------------------|-------------|
| **Parameter: Condition** | Custom formula |
| **Parameter: Type of formula** | Custom single |
| **Parameter: Condition** | ISMISMATCHED(datetime, ['Datetime','yy-mm-dd hh:mm:ss','yyyy*mm*dd*HH:MM:SS']) |
| **Parameter: Action** | Delete matching rows |

When the Datetime values are consistently formatted, you can use the following transformations to perform conversions. The following tranformation converts the values from UTC to US/Eastern time zone:

| Transformation Name | New formula |
|---------------------|-------------|
| **Parameter: Formula type** | Single row formula |

| Parameter: Formula | CONVERTFROMUTC(datetime, 'US\/Eastern') |
|---|---|
| Parameter: New column name | 'datetimeUTC2Eastern' |

This transformation now assumes that the date values are in US/Pacific time zone and converts them to UTC:

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | CONVERTTOUTC(datetime, 'US\/Pacific') |
| Parameter: New column name | 'datetimePacific2UTC' |

The final transformation converts the date time values between arbitrary time zones. In this case, the values are assumed to be in US/Alaska time zone and are converted to US/Hawaii time zone:

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | CONVERTTIMEZONE(datetime, 'US\/Alaska', 'US\/Hawaii') |
| Parameter: New column name | 'datetimeAlaska2Hawaii' |

**Results:**

| row | datetime | datetimeAlaska2Hawaii | datetimePacific2UTC | datetimeUTC2Eastern |
|---|---|---|---|---|
| 2 | 2020-03-15 00:00:00 | 2020-03-14 22:00:00 | 2020-03-15 07:00:00 | 2020-03-14 20:00:00 |
| 4 | 2020-03-15 01:02:03 | 2020-03-14 23:02:03 | 2020-03-15 08:02:03 | 2020-03-14 21:02:03 |
| 5 | 2020-03-15 04:02:03 | 2020-03-15 02:02:03 | 2020-03-15 11:02:03 | 2020-03-15 00:02:03 |
| 6 | 2020-03-15 08:02:03 | 2020-03-15 06:02:03 | 2020-03-15 15:02:03 | 2020-03-15 04:02:03 |
| 7 | 2020-03-15 12:02:03 | 2020-03-15 10:02:03 | 2020-03-15 19:02:03 | 2020-03-15 08:02:03 |
| 8 | 2020-03-15 16:02:03 | 2020-03-15 14:02:03 | 2020-03-15 23:02:03 | 2020-03-15 12:02:03 |
| 9 | 2020-03-15 20:02:03 | 2020-03-15 18:02:03 | 2020-03-16 03:02:03 | 2020-03-15 16:02:03 |
| 10 | 2020-03-15 23:02:03 | 2020-03-15 21:02:03 | 2020-03-16 06:02:03 | 2020-03-15 19:02:03 |

# MINDATEIF Function

**Contents:**

Returns the minimum Datetime value of rows in each group that meet a specific condition. Set of values must valid Datetime values.

---

> **NOTE:** When added to a transformation, this function is applied to the current sample. If you change your sample or run the job, the computed values for this function are updated. Transformations that change the number of rows in subsequent recipe steps do not affect the values computed for this step.

To calculate the minimum Datetime value of rows without conditionals, use the `MINDATE` function. See *MINDATE Function*.

**Wrangle vs. SQL:** This function is part of Wrangle , a proprietary data transformation language. Wrangle is not SQL. For more information, see *Wrangle Language*.

## Basic Usage

```
mindateif(incidentDate, serverFailure == &apos;true&apos;)
```

**Output:** Returns the minimum date from the `incidentDate` column when the value in the `serverFailure` column is `true`.

## Syntax and Arguments

```
mindateif(col_ref, test_expression) [group:group_col_ref] [limit:limit_count]
```

| Argument | Required? | Data Type | Description |
|----------|-----------|-----------|-------------|
| col_ref | Y | string | Reference to the column you wish to evaluate. Values in the columns must be valid Date values. |
| test_expression | Y | string | Expression that is evaluated. Must resolve to `true` or `false` |

For more information on syntax standards, see *Language Documentation Syntax Notes*.

For more information on the `group` and `limit` parameters, see *Pivot Transform*.

**col_ref**

Name of the column whose values you wish to use in the calculation. Inputs must be Datetime values.

**Usage Notes:**

| Required? | Data Type | Example Value |
|---|---|---|
| Yes | String that corresponds to the name of the column | `myDates` |

**test_expression**

This parameter contains the expression to evaluate. This expression must resolve to a Boolean (`true` or `false`) value.

**Usage Notes:**

| Required? | Data Type | Example Value |
|---|---|---|
| Yes | String expression that evaluates to `true` or `false` | `(LastName == 'Mouse' && FirstName == 'Mickey')` |

## Examples

> **Tip:** For additional examples, see *Common Tasks*.

### Example - DATEIF Functions

This example illustrates how you can apply conditionals to calculate minimum, maximum, and most common date values.

**Functions**:

| Item | Description |
|---|---|
| MINDATEIF Function | Returns the minimum Datetime value of rows in each group that meet a specific condition. Set of values must valid Datetime values. |
| MAXDATEIF Function | Returns the maximum Datetime value of rows in each group that meet a specific condition. Set of values must valid Datetime values. |
| MODEDATEIF Function | Returns the most common Datetime value of rows in each group that meet a specific condition. Set of values must valid Datetime values. |

**Source:**

Here is some example transaction data:

| Date | Product | Units | UnitCost | OrderValue |
|---|---|---|---|---|
| 3/28/2020 | ProductA | 4 | 10.00 | 40.00 |
| 3/8/2020 | ProductB | 4 | 20.00 | 80.00 |

| 3/12/2020 | ProductC | 2 | 30.00 | 60.00 |
|---|---|---|---|---|
| 3/23/2020 | ProductA | 1 | 10.00 | 10.00 |
| 3/20/2020 | ProductB | 2 | 20.00 | 40.00 |
| 3/12/2020 | ProductC | 9 | 30.00 | 270.00 |
| 3/28/2020 | ProductA | 5 | 10.00 | 50.00 |
| 3/23/2020 | ProductB | 8 | 20.00 | 160.00 |
| 3/16/2020 | ProductC | 9 | 30.00 | 270.00 |
| 3/8/2020 | ProductA | 5 | 10.00 | 50.00 |
| 3/10/2020 | ProductB | 3 | 20.00 | 60.00 |
| 3/13/2020 | ProductC | 1 | 30.00 | 30.00 |
| 3/12/2020 | ProductA | 7 | 10.00 | 70.00 |
| 3/10/2020 | ProductB | 7 | 20.00 | 140.00 |
| 3/24/2020 | ProductC | 9 | 30.00 | 270.00 |
| 3/15/2020 | ProductA | 8 | 10.00 | 80.00 |
| 3/10/2020 | ProductB | 5 | 20.00 | 100.00 |
| 3/10/2020 | ProductC | 4 | 30.00 | 120.00 |

**Transformation and Results:**

These functions are useful for asking questions about your data. In the following, you can review specific questions and see the results immediately.

**Question 1:** What is the earliest date when a $100.00 transaction occurred?

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | mindateif(Date, OrderValue > 100) |
| Parameter: New column name | 'Answers' |

**Results:** Value in Answers column: 3/10/2020

**Question 2:** What is the latest date when a $200.00 transaction occurred?

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | maxdateif(Date, OrderValue > 200) |
| Parameter: New column name | 'Answer' |

**Results:** Value in Answers column: 3/24/2020

**Question 3:** On what date did the most transactions occur this month?

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | modedateif(Date, OrderValue > 0) |
| Parameter: New column name | 'Answer' |

**Results:** Value in Answers column: 3/10/2020

# MAXDATEIF Function

**Contents:**

Returns the maximum Datetime value of rows in each group that meet a specific condition. Set of values must valid Datetime values.

> **NOTE:** When added to a transformation, this function is applied to the current sample. If you change your sample or run the job, the computed values for this function are updated. Transformations that change the number of rows in subsequent recipe steps do not affect the values computed for this step.

To calculate the maximum Datetime value of rows without conditionals, use the `MAXDATE` function. See *MAXDATE Function*.

**Wrangle vs. SQL:** This function is part of Wrangle , a proprietary data transformation language. Wrangle is not SQL. For more information, see *Wrangle Language*.

## Basic Usage

```
maxdateif(incidentDate, serverFailure == &apos;true&apos;)
```

**Output:** Returns the maximum date from the `incidentDate` column when the value in the `serverFailure` column is `true`.

## Syntax and Arguments

```
maxdateif(col_ref, test_expression) [group:group_col_ref] [limit:limit_count]
```

| Argument | Required? | Data Type | Description |
|---|---|---|---|
| col_ref | Y | string | Reference to the column you wish to evaluate. Values in the columns must be valid Date values. |
| test_expression | Y | string | Expression that is evaluated. Must resolve to `true` or `false` |

For more information on syntax standards, see *Language Documentation Syntax Notes*.

For more information on the `group` and `limit` parameters, see *Pivot Transform*.

**col_ref**

Name of the column whose values you wish to use in the calculation. Inputs must be Datetime values.

**Usage Notes:**

| Required? | Data Type | Example Value |
|---|---|---|
| Yes | String that corresponds to the name of the column | `myDates` |

**test_expression**

This parameter contains the expression to evaluate. This expression must resolve to a Boolean (`true` or `false`) value.

**Usage Notes:**

| Required? | Data Type | Example Value |
|---|---|---|
| Yes | String expression that evaluates to `true` or `false` | `(LastName == 'Mouse' && FirstName == 'Mickey')` |

## Examples

> **Tip:** For additional examples, see *Common Tasks*.

**Example - DATEIF Functions**

This example illustrates how you can apply conditionals to calculate minimum, maximum, and most common date values.

**Functions**:

| Item | Description |
|---|---|
| MINDATEIF Function | Returns the minimum Datetime value of rows in each group that meet a specific condition. Set of values must valid Datetime values. |
| MAXDATEIF Function | Returns the maximum Datetime value of rows in each group that meet a specific condition. Set of values must valid Datetime values. |
| MODEDATEIF Function | Returns the most common Datetime value of rows in each group that meet a specific condition. Set of values must valid Datetime values. |

**Source:**

Here is some example transaction data:

| Date | Product | Units | UnitCost | OrderValue |
|---|---|---|---|---|
| 3/28/2020 | ProductA | 4 | 10.00 | 40.00 |
| 3/8/2020 | ProductB | 4 | 20.00 | 80.00 |

| | | | | |
|---|---|---|---|---|
| 3/12/2020 | ProductC | 2 | 30.00 | 60.00 |
| 3/23/2020 | ProductA | 1 | 10.00 | 10.00 |
| 3/20/2020 | ProductB | 2 | 20.00 | 40.00 |
| 3/12/2020 | ProductC | 9 | 30.00 | 270.00 |
| 3/28/2020 | ProductA | 5 | 10.00 | 50.00 |
| 3/23/2020 | ProductB | 8 | 20.00 | 160.00 |
| 3/16/2020 | ProductC | 9 | 30.00 | 270.00 |
| 3/8/2020 | ProductA | 5 | 10.00 | 50.00 |
| 3/10/2020 | ProductB | 3 | 20.00 | 60.00 |
| 3/13/2020 | ProductC | 1 | 30.00 | 30.00 |
| 3/12/2020 | ProductA | 7 | 10.00 | 70.00 |
| 3/10/2020 | ProductB | 7 | 20.00 | 140.00 |
| 3/24/2020 | ProductC | 9 | 30.00 | 270.00 |
| 3/15/2020 | ProductA | 8 | 10.00 | 80.00 |
| 3/10/2020 | ProductB | 5 | 20.00 | 100.00 |
| 3/10/2020 | ProductC | 4 | 30.00 | 120.00 |

**Transformation and Results:**

These functions are useful for asking questions about your data. In the following, you can review specific questions and see the results immediately.

**Question 1:** What is the earliest date when a $100.00 transaction occurred?

| | |
|---|---|
| **Transformation Name** | `New formula` |
| **Parameter: Formula type** | `Single row formula` |
| **Parameter: Formula** | `mindateif(Date, OrderValue > 100)` |
| **Parameter: New column name** | `'Answers'` |

**Results:** Value in Answers column: 3/10/2020

**Question 2:** What is the latest date when a $200.00 transaction occurred?

| | |
|---|---|
| **Transformation Name** | `New formula` |
| **Parameter: Formula type** | `Single row formula` |
| **Parameter: Formula** | `maxdateif(Date, OrderValue > 200)` |
| **Parameter: New column name** | `'Answer'` |

**Results:** Value in Answers column: 3/24/2020

**Question 3:** On what date did the most transactions occur this month?

| | |
|---|---|
| **Transformation Name** | New formula |
| **Parameter: Formula type** | Single row formula |
| **Parameter: Formula** | modedateif(Date, OrderValue > 0) |
| **Parameter: New column name** | 'Answer' |

**Results:** Value in Answers column: 3/10/2020

# MODEDATEIF Function

**Contents:**

Returns the most common Datetime value of rows in each group that meet a specific condition. Set of values must valid Datetime values.

> **NOTE:** When added to a transformation, this function is applied to the current sample. If you change your sample or run the job, the computed values for this function are updated. Transformations that change the number of rows in subsequent recipe steps do not affect the values computed for this step.

To calculate the most common Datetime value of rows without conditionals, use the `MODEDATE` function. See *MODEDATE Function*.

**Wrangle vs. SQL:** This function is part of Wrangle , a proprietary data transformation language. Wrangle is not SQL. For more information, see *Wrangle Language*.

## Basic Usage

```
modedateif(incidentDate, serverFailure == &apos;true&apos;)
```

**Output:** Returns the most common date from the `incidentDate` column when the value in the `serverFailure` column is `true`.

## Syntax and Arguments

```
modedateif(col_ref, test_expression) [group:group_col_ref] [limit:limit_count]
```

| Argument | Required? | Data Type | Description |
|---|---|---|---|
| col_ref | Y | string | Reference to the column you wish to evaluate. Values in the columns must be valid Date values. |
| test_expression | Y | string | Expression that is evaluated. Must resolve to `true` or `false` |

For more information on syntax standards, see *Language Documentation Syntax Notes*.

For more information on the `group` and `limit` parameters, see *Pivot Transform*.

**col_ref**

Name of the column whose values you wish to use in the calculation. Inputs must be Datetime values.

**Usage Notes:**

| Required? | Data Type | Example Value |
|-----------|-----------|---------------|
| Yes | String that corresponds to the name of the column | `myDates` |

**test_expression**

This parameter contains the expression to evaluate. This expression must resolve to a Boolean (`true` or `false`) value.

**Usage Notes:**

| Required? | Data Type | Example Value |
|-----------|-----------|---------------|
| Yes | String expression that evaluates to `true` or `false` | `(LastName == 'Mouse' && FirstName == 'Mickey')` |

## Examples

> **Tip:** For additional examples, see *Common Tasks*.

### Example - DATEIF Functions

This example illustrates how you can apply conditionals to calculate minimum, maximum, and most common date values.

**Functions**:

| Item | Description |
|------|-------------|
| MINDATEIF Function | Returns the minimum Datetime value of rows in each group that meet a specific condition. Set of values must valid Datetime values. |
| MAXDATEIF Function | Returns the maximum Datetime value of rows in each group that meet a specific condition. Set of values must valid Datetime values. |
| MODEDATEIF Function | Returns the most common Datetime value of rows in each group that meet a specific condition. Set of values must valid Datetime values. |

**Source:**

Here is some example transaction data:

| Date | Product | Units | UnitCost | OrderValue |
|------|---------|-------|----------|------------|
| 3/28/2020 | ProductA | 4 | 10.00 | 40.00 |
| 3/8/2020 | ProductB | 4 | 20.00 | 80.00 |

| 3/12/2020 | ProductC | 2 | 30.00 | 60.00 |
|---|---|---|---|---|
| 3/23/2020 | ProductA | 1 | 10.00 | 10.00 |
| 3/20/2020 | ProductB | 2 | 20.00 | 40.00 |
| 3/12/2020 | ProductC | 9 | 30.00 | 270.00 |
| 3/28/2020 | ProductA | 5 | 10.00 | 50.00 |
| 3/23/2020 | ProductB | 8 | 20.00 | 160.00 |
| 3/16/2020 | ProductC | 9 | 30.00 | 270.00 |
| 3/8/2020 | ProductA | 5 | 10.00 | 50.00 |
| 3/10/2020 | ProductB | 3 | 20.00 | 60.00 |
| 3/13/2020 | ProductC | 1 | 30.00 | 30.00 |
| 3/12/2020 | ProductA | 7 | 10.00 | 70.00 |
| 3/10/2020 | ProductB | 7 | 20.00 | 140.00 |
| 3/24/2020 | ProductC | 9 | 30.00 | 270.00 |
| 3/15/2020 | ProductA | 8 | 10.00 | 80.00 |
| 3/10/2020 | ProductB | 5 | 20.00 | 100.00 |
| 3/10/2020 | ProductC | 4 | 30.00 | 120.00 |

**Transformation and Results:**

These functions are useful for asking questions about your data. In the following, you can review specific questions and see the results immediately.

**Question 1:** What is the earliest date when a $100.00 transaction occurred?

| Transformation Name | `New formula` |
|---|---|
| **Parameter: Formula type** | `Single row formula` |
| **Parameter: Formula** | `mindateif(Date, OrderValue > 100)` |
| **Parameter: New column name** | `'Answers'` |

**Results:** Value in Answers column: 3/10/2020

**Question 2:** What is the latest date when a $200.00 transaction occurred?

| Transformation Name | `New formula` |
|---|---|
| **Parameter: Formula type** | `Single row formula` |
| **Parameter: Formula** | `maxdateif(Date, OrderValue > 200)` |
| **Parameter: New column name** | `'Answer'` |

**Results:** Value in Answers column: 3/24/2020

**Question 3:** On what date did the most transactions occur this month?

| | |
|---|---|
| **Transformation Name** | New formula |
| **Parameter: Formula type** | Single row formula |
| **Parameter: Formula** | modedateif(Date, OrderValue > 0) |
| **Parameter: New column name** | 'Answer' |

**Results:** Value in Answers column: 3/10/2020

# KTHLARGESTDATE Function

**Contents:**

Extracts the ranked Datetime value from the values in a column, where `k=1` returns the maximum value. The value for `k` must be between 1 and 1000, inclusive. Inputs must be valid Datetime values.

For purposes of this calculation, two instances of the same value are treated as separate values. So, if your dataset contains three rows with column values `2020-02-15`, `2020-02-14`, and `2020-02-14`, then `KTHLARGESTDATE` returns `2020-02-14` for `k=2` and `k=3`.

When used in a `pivot` transform, the function is computed for each instance of the value specified in the `group` parameter. See *Pivot Transform*.

Input column must be Datetime type. Other values column are ignored. If a row contains a missing or null value, it is not factored into the calculation.

For a version of this function that applies to non-Datetime values, see *KTHLARGEST Function*.

**Wrangle vs. SQL:** This function is part of Wrangle , a proprietary data transformation language. Wrangle is not SQL. For more information, see *Wrangle Language*.

## Basic Usage

```
kthlargestdate(myDate, 2)
```

**Output:** Returns the second highest Datetime value from the `myDate` column.

## Syntax and Arguments

```
kthlargestdate(function_col_ref, k_integer) [ group:group_col_ref] [limit:limit_count]
```

| Argument | Required? | Data Type | Description |
|---|---|---|---|
| function_col_ref | Y | string | Name of column to which to apply the function |
| k_integer | Y | integer (positive) | The ranking of the value to extract from the source column |

For more information on the `group` and `limit` parameters, see *Pivot Transform*.

For more information on syntax standards, see *Language Documentation Syntax Notes*.

**function_col_ref**

Name of the column the values of which you want to calculate the mean. Inputs must be Datetime values.

- Literal values are not supported as inputs.
- Multiple columns and wildcards are not supported.

**Usage Notes:**

| Required? | Data Type | Example Value |
|-----------|-----------|---------------|
| Yes | String (column reference) | `transactionDate` |

**k_integer**

Integer representing the ranking of the value to extract from the source column.

> **NOTE:** The value for k must be an integer between 1 and 1,000 inclusive.

- `k=1` represents the maximum value in the column.
- If k is greater than or equal to the number of values in the column, the minimum value is returned.
- Missing and null values are not factored into the ranking of k.

**Usage Notes:**

| Required? | Data Type | Example Value |
|-----------|-----------|---------------|
| Yes | Integer (positive) | 4 |

## Examples

> **Tip:** For additional examples, see *Common Tasks*.

**Example - KTHLARGESTDATE functions**

This example illustrates how you can apply conditionals to calculate minimum, maximum, and most common date values.

**Functions**:

| Item | Description |
|------|-------------|
| KTHLARGESTDATE Function | Extracts the ranked Datetime value from the values in a column, where $k=1$ returns the maximum value. The value for k must be between 1 and 1000, inclusive. Inputs must be valid Datetime values. |
| KTHLARGESTUNIQUEDATE Function | Extracts the ranked unique Datetime value from the values in a column, where $k=1$ returns the maximum value. The value for k must be between 1 and 1000, inclusive. Inputs must be Datetime. |
| KTHLARGESTDATEIF Function | Extracts the ranked Datetime value from the values in a column, where $k=1$ returns the maximum value, when a specified condition is met. The value for k must be between 1 and 1000, inclusive. Inputs must be Datetime. |

| | |
|---|---|
| KTHLARGESTUNI QUEDATEIF Function | Extracts the ranked unique Datetime value from the values in a column, where $k=1$ returns the maximum value, when a specified condition is met. The value for $k$ must be between 1 and 1000, inclusive. Inputs must be Datetime. |

**Source:**

Here is some example transaction data:

| Date | Product | Units | UnitCost | OrderValue |
|---|---|---|---|---|
| 3/28/2020 | ProductA | 4 | 10.00 | 40.00 |
| 3/8/2020 | ProductB | 4 | 20.00 | 80.00 |
| 3/12/2020 | ProductC | 2 | 30.00 | 60.00 |
| 3/23/2020 | ProductA | 1 | 10.00 | 10.00 |
| 3/20/2020 | ProductB | 2 | 20.00 | 40.00 |
| 3/12/2020 | ProductC | 9 | 30.00 | 270.00 |
| 3/28/2020 | ProductA | 5 | 10.00 | 50.00 |
| 3/23/2020 | ProductB | 8 | 20.00 | 160.00 |
| 3/16/2020 | ProductC | 9 | 30.00 | 270.00 |
| 3/8/2020 | ProductA | 5 | 10.00 | 50.00 |
| 3/10/2020 | ProductB | 3 | 20.00 | 60.00 |
| 3/13/2020 | ProductC | 1 | 30.00 | 30.00 |
| 3/12/2020 | ProductA | 7 | 10.00 | 70.00 |
| 3/10/2020 | ProductB | 7 | 20.00 | 140.00 |
| 3/24/2020 | ProductC | 9 | 30.00 | 270.00 |
| 3/15/2020 | ProductA | 8 | 10.00 | 80.00 |
| 3/10/2020 | ProductB | 5 | 20.00 | 100.00 |
| 3/10/2020 | ProductC | 4 | 30.00 | 120.00 |

**Transformation:**

The following transformation computes the third highest date in the `Date` column:

| | |
|---|---|
| **Transformation Name** | New formula |
| **Parameter: Formula type** | Single row formula |
| **Parameter: Formula** | kthlargestdate(Date, 3) |
| **Parameter: New column name** | 'kthlargestdate' |

This transformation computes the third highest unique value in the `Date` column:

| | |
|---|---|
| **Transformation Name** | New formula |
| **Parameter: Formula type** | Single row formula |
| **Parameter: Formula** | kthlargestuniquedate(Date, 3) |

| Parameter: New column name | 'kthlargestuniquedate' |
|---|---|

Following transformation calculates the 3rd highest date value when the OrderValue > 200:

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | kthlargestdateif(Date, 3, OrderValue > 200) |
| Parameter: New column name | 'kthlargestdateif' |

Following transformation calculates the 3rd highest unique date value when the OrderValue > 200:

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | kthlargestuniquedateif(Date, 3, OrderValue > 200) |
| Parameter: New column name | 'kthlargestuniquedateif' |

**Results:**

| Date | Product | Units | UnitCost | OrderValue | kthlargestdate | kthlargestuniquedate | kthlargestdateif | kthlargestu |
|---|---|---|---|---|---|---|---|---|
| 3/28 /2020 | ProductA | 4 | 10.00 | 40.00 | 03-24-2020 | 03-23-2020 | 03-23-2020 | 03-23-2020 |
| 3/8 /2020 | ProductB | 4 | 20.00 | 80.00 | 03-24-2020 | 03-23-2020 | 03-23-2020 | 03-23-2020 |
| 3/12 /2020 | ProductC | 2 | 30.00 | 60.00 | 03-24-2020 | 03-23-2020 | 03-23-2020 | 03-23-2020 |
| 3/23 /2020 | ProductA | 1 | 10.00 | 10.00 | 03-24-2020 | 03-23-2020 | 03-23-2020 | 03-23-2020 |
| 3/20 /2020 | ProductB | 2 | 20.00 | 40.00 | 03-24-2020 | 03-23-2020 | 03-23-2020 | 03-23-2020 |
| 3/12 /2020 | ProductC | 9 | 30.00 | 270.00 | 03-24-2020 | 03-23-2020 | 03-23-2020 | 03-23-2020 |
| 3/28 /2020 | ProductA | 5 | 10.00 | 50.00 | 03-24-2020 | 03-23-2020 | 03-23-2020 | 03-23-2020 |
| 3/23 /2020 | ProductB | 8 | 20.00 | 160.00 | 03-24-2020 | 03-23-2020 | 03-23-2020 | 03-23-2020 |
| 3/16 /2020 | ProductC | 9 | 30.00 | 270.00 | 03-24-2020 | 03-23-2020 | 03-23-2020 | 03-23-2020 |
| 3/8 /2020 | ProductA | 5 | 10.00 | 50.00 | 03-24-2020 | 03-23-2020 | 03-23-2020 | 03-23-2020 |
| 3/10 /2020 | ProductB | 3 | 20.00 | 60.00 | 03-24-2020 | 03-23-2020 | 03-23-2020 | 03-23-2020 |
| 3/13 /2020 | ProductC | 1 | 30.00 | 30.00 | 03-24-2020 | 03-23-2020 | 03-23-2020 | 03-23-2020 |
| 3/12 /2020 | ProductA | 7 | 10.00 | 70.00 | 03-24-2020 | 03-23-2020 | 03-23-2020 | 03-23-2020 |
| 3/10 /2020 | ProductB | 7 | 20.00 | 140.00 | 03-24-2020 | 03-23-2020 | 03-23-2020 | 03-23-2020 |

| 3/24/2020 | ProductC | 9 | 30.00 | 270.00 | 03-24-2020 | 03-23-2020 | 03-23-2020 | 03-23-2020 |
|-----------|----------|---|-------|--------|------------|------------|------------|------------|
| 3/15/2020 | ProductA | 8 | 10.00 | 80.00 | 03-24-2020 | 03-23-2020 | 03-23-2020 | 03-23-2020 |
| 3/10/2020 | ProductB | 5 | 20.00 | 100.00 | 03-24-2020 | 03-23-2020 | 03-23-2020 | 03-23-2020 |
| 3/10/2020 | ProductC | 4 | 30.00 | 120.00 | 03-24-2020 | 03-23-2020 | 03-23-2020 | 03-23-2020 |

# KTHLARGESTUNIQUEDATE Function

**Contents:**

---

Extracts the ranked unique Datetime value from the values in a column, where `k=1` returns the maximum value. The value for `k` must be between 1 and 1000, inclusive. Inputs must be Datetime.

For purposes of this calculation, two instances of the same value are treated as the same value of `k`. if your dataset contains three rows with column values `2020-02-15`, `2020-02-14`, `2020-02-14`, and `2020-02-14`, then `KTHLARGESTDATE` returns `2020-02-14` for `k=2` and `2020-02-13` for `k=3`.

- For a non-unique version of this function, see *KTHLARGESTDATE Function*.

When used in a `pivot` transform, the function is computed for each instance of the value specified in the `group` parameter. See *Pivot Transform*.

Input column must be Datetime type. Other values column are ignored. If a row contains a missing or null value, it is not factored into the calculation.

For a version of this function that applies to non-Datetime values, see *KTHLARGESTUNIQUE Function*.

**Wrangle vs. SQL:** This function is part of Wrangle , a proprietary data transformation language. Wrangle is not SQL. For more information, see *Wrangle Language*.

## Basic Usage

```
kthlargestuniquedate(myDate, 3)
```

**Output:** Returns the third highest unique value from the `myDate` column.

## Syntax and Arguments

```
kthlargestuniquedate(function_col_ref, k_integer) [ group:group_col_ref] [limit:
limit_count]
```

| Argument | Required? | Data Type | Description |
|---|---|---|---|
| function_col_ref | Y | string | Name of column to which to apply the function |
| k_integer | Y | integer (positive) | The ranking of the unique value to extract from the source column |

For more information on the `group` and `limit` parameters, see *Pivot Transform*.

For more information on syntax standards, see *Language Documentation Syntax Notes*.

**function_col_ref**

Name of the column the values of which you want to calculate the mean. Inputs must be Datetime values.

- Literal values are not supported as inputs.
- Multiple columns and wildcards are not supported.

**Usage Notes:**

| Required? | Data Type | Example Value |
|-----------|-----------|---------------|
| Yes | String (column reference) | `transactionDate` |

**k_integer**

Integer representing the ranking of the unique value to extract from the source column. Duplicate values are treated as a single value for purposes of this function's calculation.

> **NOTE:** The value for $k$ must be an integer between 1 and 1,000 inclusive.

- `k=1` represents the maximum value in the column.
- If k is greater than or equal to the number of values in the column, the minimum value is returned.
- Missing and null values are not factored into the ranking of $k$.

**Usage Notes:**

| Required? | Data Type | Example Value |
|-----------|-----------|---------------|
| Yes | Integer (positive) | 4 |

## Examples

> **Tip:** For additional examples, see *Common Tasks*.

**Example - KTHLARGESTDATE functions**

This example illustrates how you can apply conditionals to calculate minimum, maximum, and most common date values.

**Functions**:

| Item | Description |
|------|-------------|
| KTHLARGESTDATE Function | Extracts the ranked Datetime value from the values in a column, where $k=1$ returns the maximum value. The value for $k$ must be between 1 and 1000, inclusive. Inputs must be valid Datetime values. |
| KTHLARGESTUNIQUEDATE Function | Extracts the ranked unique Datetime value from the values in a column, where $k=1$ returns the maximum value. The value for $k$ must be between 1 and 1000, inclusive. Inputs must be Datetime. |

| | |
|---|---|
| KTHLARGESTDAT EIF Function | Extracts the ranked Datetime value from the values in a column, where $k=1$ returns the maximum value, when a specified condition is met. The value for $k$ must be between 1 and 1000, inclusive. Inputs must be Datetime. |
| KTHLARGESTUNI QUEDATEIF Function | Extracts the ranked unique Datetime value from the values in a column, where $k=1$ returns the maximum value, when a specified condition is met. The value for $k$ must be between 1 and 1000, inclusive. Inputs must be Datetime. |

**Source:**

Here is some example transaction data:

| Date | Product | Units | UnitCost | OrderValue |
|---|---|---|---|---|
| 3/28/2020 | ProductA | 4 | 10.00 | 40.00 |
| 3/8/2020 | ProductB | 4 | 20.00 | 80.00 |
| 3/12/2020 | ProductC | 2 | 30.00 | 60.00 |
| 3/23/2020 | ProductA | 1 | 10.00 | 10.00 |
| 3/20/2020 | ProductB | 2 | 20.00 | 40.00 |
| 3/12/2020 | ProductC | 9 | 30.00 | 270.00 |
| 3/28/2020 | ProductA | 5 | 10.00 | 50.00 |
| 3/23/2020 | ProductB | 8 | 20.00 | 160.00 |
| 3/16/2020 | ProductC | 9 | 30.00 | 270.00 |
| 3/8/2020 | ProductA | 5 | 10.00 | 50.00 |
| 3/10/2020 | ProductB | 3 | 20.00 | 60.00 |
| 3/13/2020 | ProductC | 1 | 30.00 | 30.00 |
| 3/12/2020 | ProductA | 7 | 10.00 | 70.00 |
| 3/10/2020 | ProductB | 7 | 20.00 | 140.00 |
| 3/24/2020 | ProductC | 9 | 30.00 | 270.00 |
| 3/15/2020 | ProductA | 8 | 10.00 | 80.00 |
| 3/10/2020 | ProductB | 5 | 20.00 | 100.00 |
| 3/10/2020 | ProductC | 4 | 30.00 | 120.00 |

**Transformation:**

The following transformation computes the third highest date in the `Date` column:

| | |
|---|---|
| **Transformation Name** | New formula |
| **Parameter: Formula type** | Single row formula |
| **Parameter: Formula** | kthlargestdate(Date, 3) |
| **Parameter: New column name** | 'kthlargestdate' |

This transformation computes the third highest unique value in the `Date` column:

| | |
|---|---|
| **Transformation Name** | New formula |
| **Parameter: Formula type** | Single row formula |

| Parameter: Formula | kthlargestuniquedate(Date, 3) |
|---|---|
| Parameter: New column name | 'kthlargestuniquedate' |

Following transformation calculates the 3rd highest date value when the OrderValue > 200:

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | kthlargestdateif(Date, 3, OrderValue > 200) |
| Parameter: New column name | 'kthlargestdateif' |

Following transformation calculates the 3rd highest unique date value when the OrderValue > 200:

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | kthlargestuniquedateif(Date, 3, OrderValue > 200) |
| Parameter: New column name | 'kthlargestuniquedateif' |

**Results:**

| Date | Product | Units | UnitCost | OrderValue | kthlargestdate | kthlargestuniquedate | kthlargestdateif | kthlargestu |
|---|---|---|---|---|---|---|---|---|
| 3/28 /2020 | ProductA | 4 | 10.00 | 40.00 | 03-24-2020 | 03-23-2020 | 03-23-2020 | 03-23-2020 |
| 3/8 /2020 | ProductB | 4 | 20.00 | 80.00 | 03-24-2020 | 03-23-2020 | 03-23-2020 | 03-23-2020 |
| 3/12 /2020 | ProductC | 2 | 30.00 | 60.00 | 03-24-2020 | 03-23-2020 | 03-23-2020 | 03-23-2020 |
| 3/23 /2020 | ProductA | 1 | 10.00 | 10.00 | 03-24-2020 | 03-23-2020 | 03-23-2020 | 03-23-2020 |
| 3/20 /2020 | ProductB | 2 | 20.00 | 40.00 | 03-24-2020 | 03-23-2020 | 03-23-2020 | 03-23-2020 |
| 3/12 /2020 | ProductC | 9 | 30.00 | 270.00 | 03-24-2020 | 03-23-2020 | 03-23-2020 | 03-23-2020 |
| 3/28 /2020 | ProductA | 5 | 10.00 | 50.00 | 03-24-2020 | 03-23-2020 | 03-23-2020 | 03-23-2020 |
| 3/23 /2020 | ProductB | 8 | 20.00 | 160.00 | 03-24-2020 | 03-23-2020 | 03-23-2020 | 03-23-2020 |
| 3/16 /2020 | ProductC | 9 | 30.00 | 270.00 | 03-24-2020 | 03-23-2020 | 03-23-2020 | 03-23-2020 |
| 3/8 /2020 | ProductA | 5 | 10.00 | 50.00 | 03-24-2020 | 03-23-2020 | 03-23-2020 | 03-23-2020 |
| 3/10 /2020 | ProductB | 3 | 20.00 | 60.00 | 03-24-2020 | 03-23-2020 | 03-23-2020 | 03-23-2020 |
| 3/13 /2020 | ProductC | 1 | 30.00 | 30.00 | 03-24-2020 | 03-23-2020 | 03-23-2020 | 03-23-2020 |
| 3/12 /2020 | ProductA | 7 | 10.00 | 70.00 | 03-24-2020 | 03-23-2020 | 03-23-2020 | 03-23-2020 |

| 3/10 /2020 | ProductB | 7 | 20.00 | 140.00 | 03-24-2020 | 03-23-2020 | 03-23-2020 | 03-23-2020 |
|---|---|---|---|---|---|---|---|---|
| 3/24 /2020 | ProductC | 9 | 30.00 | 270.00 | 03-24-2020 | 03-23-2020 | 03-23-2020 | 03-23-2020 |
| 3/15 /2020 | ProductA | 8 | 10.00 | 80.00 | 03-24-2020 | 03-23-2020 | 03-23-2020 | 03-23-2020 |
| 3/10 /2020 | ProductB | 5 | 20.00 | 100.00 | 03-24-2020 | 03-23-2020 | 03-23-2020 | 03-23-2020 |
| 3/10 /2020 | ProductC | 4 | 30.00 | 120.00 | 03-24-2020 | 03-23-2020 | 03-23-2020 | 03-23-2020 |

# KTHLARGESTUNIQUEDATEIF Function

**Contents:**

Extracts the ranked unique Datetime value from the values in a column, where `k=1` returns the maximum value, when a specified condition is met. The value for `k` must be between 1 and 1000, inclusive. Inputs must be Datetime.

`KTHLARGESTUNIQUEDATEIF` calculations are filtered by a conditional applied to the group.

For purposes of this calculation, two instances of the same value are treated as the same value of `k`. So, if your dataset contains four rows with column values `2020-02-15`, `2020-02-14`, `2020-02-14`, and `2020-02-13`, then `KTHLARGESTUNIQUEDATEIF` returns `2020-02-14` for `k=2` and `2020-02-13` for `k=3`.

Input column must be of Datetime type. Other values column are ignored. If a row contains a missing or null value, it is not factored into the calculation.

> **NOTE:** When added to a transformation, this function is applied to the current sample. If you change your sample or run the job, the computed values for this function are updated. Transformations that change the number of rows in subsequent recipe steps do not affect the values computed for this step.

To perform a simple kth largest unique calculation on Datetime values without conditionals, use the `KTHLARGESTUNIQUEDATE` function. See *KTHLARGESTUNIQUEDATE Function*.

For a version of this function that applies to non-Datetime values, see *KTHLARGESTUNIQUE Function*.

**Wrangle vs. SQL:** This function is part of Wrangle , a proprietary data transformation language. Wrangle is not SQL. For more information, see *Wrangle Language*.

## Basic Usage

```
kthlargestuniquedateif(transDate, 2, salesPerson == &apos;jsmith&apos;)
```

**Output:** Returns the secondmost recent unique Date (rank=2) from the `transDate` column when the `salesPerson` value is `jsmith`.

## Syntax and Arguments

```
kthlargestuniquedateif(col_ref, limit, test_expression) [group:group_col_ref] [limit:
limit_count]
```

| Argument | Required? | Data Type | Description |
|----------|-----------|-----------|-------------|

| col_ref | Y | string | Reference to the column you wish to evaluate. |
|---|---|---|---|
| k_integer | Y | integer | The ranking of the value to extract from the source column |
| test_expression | Y | string | Expression that is evaluated. Must resolve to `true` or `false` |

For more information on syntax standards, see *Language Documentation Syntax Notes*.

For more information on the `group` and `limit` parameter, see *Pivot Transform*.

### col_ref

Name of the column whose values you wish to use in the calculation. Inputs must be Datetime values.

**Usage Notes:**

| Required? | Data Type | Example Value |
|---|---|---|
| Yes | String that corresponds to the name of the column | `transactionDate` |

### k_integer

Integer representing the unique ranking of the value to extract from the source column.

> **NOTE:** The value for `k` must be an integer between 1 and 1,000 inclusive.

- `k=1` represents the maximum value in the column.
- If k is greater than or equal to the number of values in the column, the minimum value is returned.
- Missing and null values are not factored into the ranking of `k`.

### test_expression

This parameter contains the expression to evaluate. This expression must resolve to a Boolean (`true` or `false`) value.

**Usage Notes:**

| Required? | Data Type | Example Value |
|---|---|---|
| Yes | String expression that evaluates to `true` or `false` | `(LastName == 'Mouse' && FirstName == 'Mickey')` |

## Examples

> **Tip:** For additional examples, see *Common Tasks*.

### Example - KTHLARGESTDATE functions

This example illustrates how you can apply conditionals to calculate minimum, maximum, and most common date values.

**Functions**:

| Item | Description |
|---|---|
| KTHLARGESTDATE Function | Extracts the ranked Datetime value from the values in a column, where $k=1$ returns the maximum value. The value for $k$ must be between 1 and 1000, inclusive. Inputs must be valid Datetime values. |
| KTHLARGESTUNIQUEDATE Function | Extracts the ranked unique Datetime value from the values in a column, where $k=1$ returns the maximum value. The value for $k$ must be between 1 and 1000, inclusive. Inputs must be Datetime. |
| KTHLARGESTDATEIF Function | Extracts the ranked Datetime value from the values in a column, where $k=1$ returns the maximum value, when a specified condition is met. The value for $k$ must be between 1 and 1000, inclusive. Inputs must be Datetime. |
| KTHLARGESTUNIQUEDATEIF Function | Extracts the ranked unique Datetime value from the values in a column, where $k=1$ returns the maximum value, when a specified condition is met. The value for $k$ must be between 1 and 1000, inclusive. Inputs must be Datetime. |

**Source:**

Here is some example transaction data:

| Date | Product | Units | UnitCost | OrderValue |
|---|---|---|---|---|
| 3/28/2020 | ProductA | 4 | 10.00 | 40.00 |
| 3/8/2020 | ProductB | 4 | 20.00 | 80.00 |
| 3/12/2020 | ProductC | 2 | 30.00 | 60.00 |
| 3/23/2020 | ProductA | 1 | 10.00 | 10.00 |
| 3/20/2020 | ProductB | 2 | 20.00 | 40.00 |
| 3/12/2020 | ProductC | 9 | 30.00 | 270.00 |
| 3/28/2020 | ProductA | 5 | 10.00 | 50.00 |
| 3/23/2020 | ProductB | 8 | 20.00 | 160.00 |
| 3/16/2020 | ProductC | 9 | 30.00 | 270.00 |
| 3/8/2020 | ProductA | 5 | 10.00 | 50.00 |
| 3/10/2020 | ProductB | 3 | 20.00 | 60.00 |
| 3/13/2020 | ProductC | 1 | 30.00 | 30.00 |
| 3/12/2020 | ProductA | 7 | 10.00 | 70.00 |
| 3/10/2020 | ProductB | 7 | 20.00 | 140.00 |
| 3/24/2020 | ProductC | 9 | 30.00 | 270.00 |
| 3/15/2020 | ProductA | 8 | 10.00 | 80.00 |
| 3/10/2020 | ProductB | 5 | 20.00 | 100.00 |
| 3/10/2020 | ProductC | 4 | 30.00 | 120.00 |

**Transformation:**

The following transformation computes the third highest date in the `Date` column:

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | kthlargestdate(Date, 3) |

| Parameter: New column name | 'kthlargestdate' |
|---|---|

This transformation computes the third highest unique value in the `Date` column:

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | kthlargestuniquedate(Date, 3) |
| Parameter: New column name | 'kthlargestuniquedate' |

Following transformation calculates the 3rd highest date value when the OrderValue > 200:

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | kthlargestdateif(Date, 3, OrderValue > 200) |
| Parameter: New column name | 'kthlargestdateif' |

Following transformation calculates the 3rd highest unique date value when the OrderValue > 200:

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | kthlargestuniquedateif(Date, 3, OrderValue > 200) |
| Parameter: New column name | 'kthlargestuniquedateif' |

**Results:**

| Date | Product | Units | UnitCost | OrderValue | kthlargestdate | kthlargestuniquedate | kthlargestdateif | kthlargestu |
|---|---|---|---|---|---|---|---|---|
| 3/28 /2020 | ProductA | 4 | 10.00 | 40.00 | 03-24-2020 | 03-23-2020 | 03-23-2020 | 03-23-2020 |
| 3/8 /2020 | ProductB | 4 | 20.00 | 80.00 | 03-24-2020 | 03-23-2020 | 03-23-2020 | 03-23-2020 |
| 3/12 /2020 | ProductC | 2 | 30.00 | 60.00 | 03-24-2020 | 03-23-2020 | 03-23-2020 | 03-23-2020 |
| 3/23 /2020 | ProductA | 1 | 10.00 | 10.00 | 03-24-2020 | 03-23-2020 | 03-23-2020 | 03-23-2020 |
| 3/20 /2020 | ProductB | 2 | 20.00 | 40.00 | 03-24-2020 | 03-23-2020 | 03-23-2020 | 03-23-2020 |
| 3/12 /2020 | ProductC | 9 | 30.00 | 270.00 | 03-24-2020 | 03-23-2020 | 03-23-2020 | 03-23-2020 |
| 3/28 /2020 | ProductA | 5 | 10.00 | 50.00 | 03-24-2020 | 03-23-2020 | 03-23-2020 | 03-23-2020 |
| 3/23 /2020 | ProductB | 8 | 20.00 | 160.00 | 03-24-2020 | 03-23-2020 | 03-23-2020 | 03-23-2020 |
| 3/16 /2020 | ProductC | 9 | 30.00 | 270.00 | 03-24-2020 | 03-23-2020 | 03-23-2020 | 03-23-2020 |

| 3/8/2020 | ProductA | 5 | 10.00 | 50.00 | 03-24-2020 | 03-23-2020 | 03-23-2020 | 03-23-2020 |
|---|---|---|---|---|---|---|---|---|
| 3/10/2020 | ProductB | 3 | 20.00 | 60.00 | 03-24-2020 | 03-23-2020 | 03-23-2020 | 03-23-2020 |
| 3/13/2020 | ProductC | 1 | 30.00 | 30.00 | 03-24-2020 | 03-23-2020 | 03-23-2020 | 03-23-2020 |
| 3/12/2020 | ProductA | 7 | 10.00 | 70.00 | 03-24-2020 | 03-23-2020 | 03-23-2020 | 03-23-2020 |
| 3/10/2020 | ProductB | 7 | 20.00 | 140.00 | 03-24-2020 | 03-23-2020 | 03-23-2020 | 03-23-2020 |
| 3/24/2020 | ProductC | 9 | 30.00 | 270.00 | 03-24-2020 | 03-23-2020 | 03-23-2020 | 03-23-2020 |
| 3/15/2020 | ProductA | 8 | 10.00 | 80.00 | 03-24-2020 | 03-23-2020 | 03-23-2020 | 03-23-2020 |
| 3/10/2020 | ProductB | 5 | 20.00 | 100.00 | 03-24-2020 | 03-23-2020 | 03-23-2020 | 03-23-2020 |
| 3/10/2020 | ProductC | 4 | 30.00 | 120.00 | 03-24-2020 | 03-23-2020 | 03-23-2020 | 03-23-2020 |

# KTHLARGESTDATEIF Function

**Contents:**

---

Extracts the ranked Datetime value from the values in a column, where `k=1` returns the maximum value, when a specified condition is met. The value for `k` must be between 1 and 1000, inclusive. Inputs must be Datetime.

`KTHLARGESTDATEIF` calculations are filtered by a conditional applied to the group.

For purposes of this calculation, two instances of the same value are treated as the same value of `k`. So, if your dataset contains three rows with column values `2020-02-15`, `2020-02-14`, and `2020-02-14`, then `KTHLARGESTDATEIF` returns `2020-02-14` for `k=2` and `2020-02-14` for `k=3`.

Input column must be of Datetime type. Other values column are ignored. If a row contains a missing or null value, it is not factored into the calculation.

> **NOTE:** When added to a transformation, this function is applied to the current sample. If you change your sample or run the job, the computed values for this function are updated. Transformations that change the number of rows in subsequent recipe steps do not affect the values computed for this step.

To perform a simple kth largest calculation on Datetime values without conditionals, use the `KTHLARGESTDATE` function. See *KTHLARGESTDATE Function*.

For a version of this function that applies to non-Datetime values, see *KTHLARGESTIF Function*.

**Wrangle vs. SQL:** This function is part of Wrangle , a proprietary data transformation language. Wrangle is not SQL. For more information, see *Wrangle Language*.

## Basic Usage

```
kthlargestdateif(transDate, 2, salesPerson == &apos;jsmith&apos;)
```

**Output:** Returns the secondmost recent Date (rank=2) from the `transDate` column when the `salesPerson` value is `jsmith`.

## Syntax and Arguments

```
kthlargestdateif(col_ref, limit, test_expression) [group:group_col_ref] [limit:limit_count]
```

| Argument | Required? | Data Type | Description |
|---|---|---|---|
| col_ref | Y | string | Reference to the column you wish to evaluate. |

| k_integer | Y | integer | The ranking of the value to extract from the source column |
|---|---|---|---|
| test_expression | Y | string | Expression that is evaluated. Must resolve to `true` or `false` |

For more information on syntax standards, see *Language Documentation Syntax Notes*.

For more information on the `group` and `limit` parameter, see *Pivot Transform*.

**col_ref**

Name of the column whose values you wish to use in the calculation. Inputs must be Datetime values.

**Usage Notes:**

| Required? | Data Type | Example Value |
|---|---|---|
| Yes | String that corresponds to the name of the column | `transactionDate` |

**k_integer**

Integer representing the ranking of the value to extract from the source column.

> **NOTE:** The value for `k` must be an integer between 1 and 1,000 inclusive.

- `k=1` represents the maximum value in the column.
- If k is greater than or equal to the number of values in the column, the minimum value is returned.
- Missing and null values are not factored into the ranking of `k`.

**test_expression**

This parameter contains the expression to evaluate. This expression must resolve to a Boolean (`true` or `false`) value.

**Usage Notes:**

| Required? | Data Type | Example Value |
|---|---|---|
| Yes | String expression that evaluates to `true` or `false` | `(LastName == 'Mouse' && FirstName == 'Mickey')` |

Examples

> **Tip:** For additional examples, see *Common Tasks*.

**Example - KTHLARGESTDATE functions**

This example illustrates how you can apply conditionals to calculate minimum, maximum, and most common date values.

**Functions**:

| Item | Description |
|------|-------------|
| KTHLARGESTDAT E Function | Extracts the ranked Datetime value from the values in a column, where $k=1$ returns the maximum value. The value for $k$ must be between 1 and 1000, inclusive. Inputs must be valid Datetime values. |
| KTHLARGESTUNI QUEDATE Function | Extracts the ranked unique Datetime value from the values in a column, where $k=1$ returns the maximum value. The value for $k$ must be between 1 and 1000, inclusive. Inputs must be Datetime. |
| KTHLARGESTDAT EIF Function | Extracts the ranked Datetime value from the values in a column, where $k=1$ returns the maximum value, when a specified condition is met. The value for $k$ must be between 1 and 1000, inclusive. Inputs must be Datetime. |
| KTHLARGESTUNI QUEDATEIF Function | Extracts the ranked unique Datetime value from the values in a column, where $k=1$ returns the maximum value, when a specified condition is met. The value for $k$ must be between 1 and 1000, inclusive. Inputs must be Datetime. |

**Source:**

Here is some example transaction data:

| Date | Product | Units | UnitCost | OrderValue |
|------|---------|-------|----------|------------|
| 3/28/2020 | ProductA | 4 | 10.00 | 40.00 |
| 3/8/2020 | ProductB | 4 | 20.00 | 80.00 |
| 3/12/2020 | ProductC | 2 | 30.00 | 60.00 |
| 3/23/2020 | ProductA | 1 | 10.00 | 10.00 |
| 3/20/2020 | ProductB | 2 | 20.00 | 40.00 |
| 3/12/2020 | ProductC | 9 | 30.00 | 270.00 |
| 3/28/2020 | ProductA | 5 | 10.00 | 50.00 |
| 3/23/2020 | ProductB | 8 | 20.00 | 160.00 |
| 3/16/2020 | ProductC | 9 | 30.00 | 270.00 |
| 3/8/2020 | ProductA | 5 | 10.00 | 50.00 |
| 3/10/2020 | ProductB | 3 | 20.00 | 60.00 |
| 3/13/2020 | ProductC | 1 | 30.00 | 30.00 |
| 3/12/2020 | ProductA | 7 | 10.00 | 70.00 |
| 3/10/2020 | ProductB | 7 | 20.00 | 140.00 |
| 3/24/2020 | ProductC | 9 | 30.00 | 270.00 |
| 3/15/2020 | ProductA | 8 | 10.00 | 80.00 |
| 3/10/2020 | ProductB | 5 | 20.00 | 100.00 |
| 3/10/2020 | ProductC | 4 | 30.00 | 120.00 |

**Transformation:**

The following transformation computes the third highest date in the `Date` column:

| Transformation Name | New formula |
|---------------------|-------------|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | kthlargestdate(Date, 3) |

| Parameter: New column name | 'kthlargestdate' |
|---|---|

This transformation computes the third highest unique value in the Date column:

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | kthlargestuniquedate(Date, 3) |
| Parameter: New column name | 'kthlargestuniquedate' |

Following transformation calculates the 3rd highest date value when the OrderValue > 200:

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | kthlargestdateif(Date, 3, OrderValue > 200) |
| Parameter: New column name | 'kthlargestdateif' |

Following transformation calculates the 3rd highest unique date value when the OrderValue > 200:

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | kthlargestuniquedateif(Date, 3, OrderValue > 200) |
| Parameter: New column name | 'kthlargestuniquedateif' |

**Results:**

| Date | Product | Units | UnitCost | OrderValue | kthlargestdate | kthlargestuniquedate | kthlargestdateif | kthlargestu |
|---|---|---|---|---|---|---|---|---|
| 3/28 /2020 | ProductA | 4 | 10.00 | 40.00 | 03-24-2020 | 03-23-2020 | 03-23-2020 | 03-23-2020 |
| 3/8 /2020 | ProductB | 4 | 20.00 | 80.00 | 03-24-2020 | 03-23-2020 | 03-23-2020 | 03-23-2020 |
| 3/12 /2020 | ProductC | 2 | 30.00 | 60.00 | 03-24-2020 | 03-23-2020 | 03-23-2020 | 03-23-2020 |
| 3/23 /2020 | ProductA | 1 | 10.00 | 10.00 | 03-24-2020 | 03-23-2020 | 03-23-2020 | 03-23-2020 |
| 3/20 /2020 | ProductB | 2 | 20.00 | 40.00 | 03-24-2020 | 03-23-2020 | 03-23-2020 | 03-23-2020 |
| 3/12 /2020 | ProductC | 9 | 30.00 | 270.00 | 03-24-2020 | 03-23-2020 | 03-23-2020 | 03-23-2020 |
| 3/28 /2020 | ProductA | 5 | 10.00 | 50.00 | 03-24-2020 | 03-23-2020 | 03-23-2020 | 03-23-2020 |
| 3/23 /2020 | ProductB | 8 | 20.00 | 160.00 | 03-24-2020 | 03-23-2020 | 03-23-2020 | 03-23-2020 |
| 3/16 /2020 | ProductC | 9 | 30.00 | 270.00 | 03-24-2020 | 03-23-2020 | 03-23-2020 | 03-23-2020 |

| 3/8/2020 | ProductA | 5 | 10.00 | 50.00 | 03-24-2020 | 03-23-2020 | 03-23-2020 | 03-23-2020 |
|---|---|---|---|---|---|---|---|---|
| 3/10/2020 | ProductB | 3 | 20.00 | 60.00 | 03-24-2020 | 03-23-2020 | 03-23-2020 | 03-23-2020 |
| 3/13/2020 | ProductC | 1 | 30.00 | 30.00 | 03-24-2020 | 03-23-2020 | 03-23-2020 | 03-23-2020 |
| 3/12/2020 | ProductA | 7 | 10.00 | 70.00 | 03-24-2020 | 03-23-2020 | 03-23-2020 | 03-23-2020 |
| 3/10/2020 | ProductB | 7 | 20.00 | 140.00 | 03-24-2020 | 03-23-2020 | 03-23-2020 | 03-23-2020 |
| 3/24/2020 | ProductC | 9 | 30.00 | 270.00 | 03-24-2020 | 03-23-2020 | 03-23-2020 | 03-23-2020 |
| 3/15/2020 | ProductA | 8 | 10.00 | 80.00 | 03-24-2020 | 03-23-2020 | 03-23-2020 | 03-23-2020 |
| 3/10/2020 | ProductB | 5 | 20.00 | 100.00 | 03-24-2020 | 03-23-2020 | 03-23-2020 | 03-23-2020 |
| 3/10/2020 | ProductC | 4 | 30.00 | 120.00 | 03-24-2020 | 03-23-2020 | 03-23-2020 | 03-23-2020 |

# SERIALNUMBER Function

Generates a serial date number from a valid date value.

- Serial date number values begin with January 1, 1900.
- Source values can be a valid date or output of a function that generates a valid date.

**Wrangle vs. SQL:** This function is part of Wrangle , a proprietary data transformation language. Wrangle is not SQL. For more information, see *Wrangle Language*.

## Basic Usage

**Integer literal values:**

```
serialnumber(date(2015,02,15))
```

**Output:** Returns the serial date number for February 15, 2015.

**Column reference values:**

```
serialnumber(date(myYear, myMonth, myDay))
```

**Output:** Returns serial date number based on three input columns, which contain valid values for day, month, and year in the calendar.

## Syntax and Arguments

```
serialnumber(date_col)
```

| Argument | Required? | Data Type | Description |
|----------|-----------|-----------|-------------|
| date_col | Y | Datetime | Date literal, name of column containing valid date values, or function returning a valid date value. |

For more information on syntax standards, see *Language Documentation Syntax Notes*.

**date_col**

Date literal, column containing date values, or function returning date values. Time values in the input value are not used.

- Missing values for this function in the source data result in missing values in the output.
- Multiple columns and wildcards are not supported.

**Usage Notes:**

| Required? | Data Type | Example Value |
|-----------|-----------|---------------|
| Yes | Datetime (literal,column reference, or function) | DATE(2020,01,02) |

## Examples

**Example - serial date numbers**

**Source:**

| eventId | Year | Month | Day |
|---------|------|-------|-----|
| e001 | 2021 | 01 | 01 |
| e002 | 2021 | 01 | 02 |
| e003 | 2021 | 01 | 03 |
| e004 | 2021 | 01 | 04 |
| e005 | 2021 | 01 | 05 |

**Transformation:**

| | |
|---|---|
| **Transformation Name** | `New formula` |
| **Parameter: Formula type** | `Single row formula` |
| **Parameter: Formula** | `serialnumber(date(Year, Month, Day))` |
| **Parameter: New column name** | `'datSerialNumber'` |

**Results:**

| eventId | Year | Month | Day | datSerialNumber |
|---------|------|-------|-----|-----------------|
| e001 | 2021 | 01 | 01 | 44197 |
| e002 | 2021 | 01 | 02 | 44198 |
| e003 | 2021 | 01 | 03 | 44199 |
| e004 | 2021 | 01 | 04 | 44200 |
| e005 | 2021 | 01 | 05 | 44201 |

# String Functions

A string is any sequence of characters. The following values can all be treated as strings:

| myStrings |
|-----------|
| Jack Jones |
| 123 |
| true |
| NY |

> **Tip:** Any value can be a string. So, you can change the data type of any column to string to use these functions. In some cases, however, it is easier to use other functions that are designed for the original data type. Make sure you switch the type back after completing your string manipulation.

String functions perform operations on text data. Based on column or string literal inputs, these functions render outputs that provide a subset of the input, convert to a different format or case, or compute metrics about the input. These functions are very useful for manipulating strings.

- Some string functions utilize Patterns and regular expressions to identify patterns to match.
- Depending on the type of string function, some types of string literals or patterns might not be supported.

For more information on these expressions, see *Text Matching*.

# CHAR Function

Generates the Unicode character corresponding to an inputted Integer value.

**Unicode** is a digital standard for the consistent encoding of the world's writing systems, so that representation of character sets is consistent around the world.

- The first 256 Unicode characters (0, 255) correspond to the ASCII character set.
- Input values for the `CHAR` function should be of integer type. Decimal type column data can be used as input. However, if the data contains digits to the right of the decimal point, the `CHAR` function returns a missing value.
- If the function cannot evaluate the numeric data, a null value is returned.

**Wrangle vs. SQL:** This function is part of Wrangle , a proprietary data transformation language. Wrangle is not SQL. For more information, see *Wrangle Language*.

## Basic Usage

**Column reference example:**

```
char(MyCharIndex)
```

**Output:** Returns the Unicode value for the number in the `MyCharIndex` column.

**String literal example:**

```
char(65)
```

**Output:** Returns the string: `A`.

## Syntax and Arguments

```
char(index_value)
```

| Argument | Required? | Data Type | Description |
|----------|-----------|-----------|-------------|
| index_value | Y | integer (positive) | Unicode index value of the character |

For more information on syntax standards, see *Language Documentation Syntax Notes*.

**index_value**

Unicode index value of the character to generate or match.

- The Unicode character set contains up 1,114,112 characters. Most uses rely on the first 10,000 characters.
- Value must be less than `end_index`.

**Usage Notes:**

| Required? | Data Type | Example Value |
|-----------|-----------|---------------|

| | | |
|---|---|---|
| Yes | Integer (non-negative) | 65 |

## Examples

> **Tip:** For additional examples, see *Common Tasks*.

### Example - char and unicode functions

In this example, you can see how the `CHAR` function can be used to convert numeric index values to Unicode characters, and the `UNICODE` function can be used to convert characters back to numeric values.

**Functions:**

| Item | Description |
|---|---|
| CHAR Function | Generates the Unicode character corresponding to an inputted Integer value. |
| UNICODE Function | Generates the Unicode index value for the first character of the input string. |

**Source:**

The following column contains some source index values:

| index |
|---|
| 1 |
| 33 |
| 33.5 |
| 34 |
| 48 |
| 57 |
| 65 |
| 90 |
| 97 |
| 121 |
| 254 |
| 255 |
| 256 |
| 257 |
| 9998 |
| 9999 |

**Transformation:**

When the above values are imported to the Transformer page, the column is typed as integer, with a single mismatched value (`33.5`). To see the corresponding Unicode characters for these characters, enter the following transformation:

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | CHAR(index) |
| Parameter: New column name | 'char_index' |

To see how these characters map back to the index values, now add the following transformation:

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | UNICODE(char_index) |
| Parameter: New column name | 'unicode_char_index' |

**Results:**

| index | char_index | unicode_char_index |
|---|---|---|
| 1 | | 1 |
| 33 | ! | 33 |
| 33.5 | | |
| 34 | " | 34 |
| 48 | 0 | 48 |
| 57 | 9 | 57 |
| 65 | A | 65 |
| 90 | Z | 90 |
| 97 | a | 97 |
| 122 | z | 122 |
| 254 | þ | 254 |
| 255 | ÿ | 255 |
| 256 | | 256 |
| 257 | | 257 |
| 9998 | | 9998 |
| 9999 | | 9999 |

Note that the floating point input value was not processed.

# UNICODE Function

Generates the Unicode index value for the first character of the input string.

- **Unicode** is a digital standard for the consistent encoding of the world's writing systems, so that representation of character sets is consistent around the world.
- The first 256 Unicode characters (0, 255) correspond to the ASCII character set.
- If the function cannot resolve a Unicode character from the first character, it returns a null value.

**Wrangle vs. SQL:** This function is part of Wrangle , a proprietary data transformation language. Wrangle is not SQL. For more information, see *Wrangle Language*.

## Basic Usage

**Column reference example:**

```
unicode(MyChar)
```

**Output:** Returns Unicode index value for the first character in the `MyChar` column.

**String literal example:**

```
unicode('A')
```

**Output:** Returns the integer `65`.

## Syntax and Arguments

```
unicode(column_string)
```

| Argument | Required? | Data Type | Description |
|----------|-----------|-----------|-------------|
| column_string | Y | string | Name of the column or string literal the Unicode value of which is generated |

For more information on syntax standards, see *Language Documentation Syntax Notes*.

**column_string**

Name of the column or string literal, the first character of which is converted to its corresponding Unicode value.

> **NOTE:** If the input string contains multiple characters, the first character is mapped to its Unicode value, and the rest are ignored.

- Missing string or column values generate missing string results.
- String constants must be quoted (`'Hello, World'`).
- Multiple columns and wildcards are not supported.

**Usage Notes:**

| Required? | Data Type | Example Value |
|-----------|-----------|---------------|
| Yes | String literal or column reference | `myColumn` |

# Examples

### Example - char and unicode functions

In this example, you can see how the CHAR function can be used to convert numeric index values to Unicode characters, and the UNICODE function can be used to convert characters back to numeric values.

### Functions:

| Item | Description |
|------|-------------|
| CHAR Function | Generates the Unicode character corresponding to an inputted Integer value. |
| UNICODE Function | Generates the Unicode index value for the first character of the input string. |

### Source:

The following column contains some source index values:

| index |
|-------|
| 1 |
| 33 |
| 33.5 |
| 34 |
| 48 |
| 57 |
| 65 |
| 90 |
| 97 |
| 121 |
| 254 |
| 255 |
| 256 |
| 257 |
| 9998 |
| 9999 |

### Transformation:

When the above values are imported to the Transformer page, the column is typed as integer, with a single mismatched value (33.5). To see the corresponding Unicode characters for these characters, enter the following transformation:

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | CHAR(index) |
| Parameter: New column name | 'char_index' |

To see how these characters map back to the index values, now add the following transformation:

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | UNICODE(char_index) |
| Parameter: New column name | 'unicode_char_index' |

**Results:**

| index | char_index | unicode_char_index |
|---|---|---|
| 1 | | 1 |
| 33 | ! | 33 |
| 33.5 | | |
| 34 | " | 34 |
| 48 | 0 | 48 |
| 57 | 9 | 57 |
| 65 | A | 65 |
| 90 | Z | 90 |
| 97 | a | 97 |
| 122 | z | 122 |
| 254 | þ | 254 |
| 255 | ÿ | 255 |
| 256 | | 256 |
| 257 | | 257 |
| 9998 | | 9998 |
| 9999 | | 9999 |

Note that the floating point input value was not processed.

# UPPER Function

All alphabetical characters in the input value are converted to uppercase in the output value.

Input can be a column reference or a string literal.

**Wrangle vs. SQL:** This function is part of Wrangle , a proprietary data transformation language. Wrangle is not SQL. For more information, see *Wrangle Language*.

## Basic Usage

**Column reference example:**

```
upper(MyName)
```

**Output:** Returns the values from the `MyName` column written in UPPER.

**String literal example:**

```
upper('Hello, World')
```

**Output:** Returns the string: `HELLO, WORLD`.

## Syntax and Arguments

```
upper(column_string)
```

| Argument | Required? | Data Type | Description |
|---|---|---|---|
| column_string | Y | string | Name of the column or string literal to be applied to the function |

For more information on syntax standards, see *Language Documentation Syntax Notes*.

### column_string

Name of the column or string constant to be converted.

- Missing string or column values generate missing string results.
- String constants must be quoted (`'Hello, World'`).
- Multiple columns and wildcards are not supported.

**Usage Notes:**

| Required? | Data Type | Example Value |
|---|---|---|
| Yes | String literal or column reference | `myColumn` |

## Examples

> **Tip:** For additional examples, see *Common Tasks.*

### Example - uppercase, lowercase, propercase functions

This example demonstrates functions that can be used to change the case of String values.

**Functions:**

| Item | Description |
|------|-------------|
| UPPER Function | All alphabetical characters in the input value are converted to uppercase in the output value. |
| LOWER Function | All alphabetical characters in the input value are converted to lowercase in the output value. |
| PROPER Function | Converts an input string to propercase. Input can be a column reference or a string literal. |

**Source:**

In the following example, you can see a number of input values in the leftmost column. The output columns are blank.

| input | uppercase | lowercase | propercase |
|-------|-----------|-----------|------------|
| AbCdEfGh IjKlMnO | | | |
| go West, young man! | | | |
| Oh, *(*$%(&! That HURT! | | | |
| A11 w0rk and n0 0play makes Jack a dull boy. | | | |

**Transformation:**

To generate uppercase, lowercase, and propercase values in the output columns, use the following transforms:

| Transformation Name | Edit column with formula |
|---------------------|--------------------------|
| Parameter: Columns | uppercase |
| Parameter: Formula | UPPER(input) |

| Transformation Name | Edit column with formula |
|---------------------|--------------------------|
| Parameter: Columns | lowercase |
| Parameter: Formula | LOWER(input) |

| Transformation Name | Edit column with formula |
|---------------------|--------------------------|
| Parameter: Columns | propercase |
| Parameter: Formula | PROPER(input) |

**Results:**

| input | uppercase | lowercase | propercase |
|---|---|---|---|
| AbCdEfGh IjKlMnO | ABCDEFGH IJKLMNO | abcdefgh ijklmno | Abcdefgh Ijklmno |
| go West, young man! | GO WEST, YOUNG MAN! | go west, young man! | Go West, Young Man! |
| Oh, *(*$%(&! That HURT! | OH, *(*$%(&! THAT HURT! | oh, *(*$%(&! that hurt! | Oh, *(*$%(&! That Hurt! |
| A11 w0rk and n0 0play makes Jack a dull boy. | A11 W0RK AND N0 0PLAY MAKES JACK A DULL BOY. | a11 w0rk and n0 0play makes jack a dull boy. | A11 W0rk And N0 0play Makes Jack A Dull Boy. |

# LOWER Function

All alphabetical characters in the input value are converted to lowercase in the output value.

Input can be a column reference or a string literal.

**Wrangle vs. SQL:** This function is part of Wrangle , a proprietary data transformation language. Wrangle is not SQL. For more information, see *Wrangle Language*.

## Basic Usage

**Column reference example:**

```
lower(MyName)
```

**Output:** Returns the string value from the `MyName` column in lowercase.

**String literal example:**

```
lower(&apos;Hello, World&apos;)
```

**Output:** The string `hello, world` is returned.

## Syntax and Arguments

```
lower(column_string)
```

| Argument | Required? | Data Type | Description |
|---|---|---|---|
| column_string | Y | string | Name of the column or string literal to be applied to the function |

For more information on syntax standards, see *Language Documentation Syntax Notes*.

**column_string**

Name of the column or string constant to be converted.

- Missing string or column values generate missing string results.
- String constants must be quoted (`'Hello, World'`).
- Multiple columns and wildcards are not supported.

**Usage Notes:**

| Required? | Data Type | Example Value |
|---|---|---|
| Yes | String literal or column reference | `myColumn` |

## Examples

> **Tip:** For additional examples, see *Common Tasks*.

## Example - uppercase, lowercase, propercase functions

This example demonstrates functions that can be used to change the case of String values.

### Functions:

| Item | Description |
|---|---|
| UPPER Function | All alphabetical characters in the input value are converted to uppercase in the output value. |
| LOWER Function | All alphabetical characters in the input value are converted to lowercase in the output value. |
| PROPER Function | Converts an input string to propercase. Input can be a column reference or a string literal. |

### Source:

In the following example, you can see a number of input values in the leftmost column. The output columns are blank.

| input | uppercase | lowercase | propercase |
|---|---|---|---|
| AbCdEfGh IjKlMnO | | | |
| go West, young man! | | | |
| Oh, *(*$%(&! That HURT! | | | |
| A11 w0rk and n0 0play makes Jack a dull boy. | | | |

### Transformation:

To generate uppercase, lowercase, and propercase values in the output columns, use the following transforms:

| Transformation Name | Edit column with formula |
|---|---|
| Parameter: Columns | uppercase |
| Parameter: Formula | UPPER(input) |

| Transformation Name | Edit column with formula |
|---|---|
| Parameter: Columns | lowercase |
| Parameter: Formula | LOWER(input) |

| Transformation Name | Edit column with formula |
|---|---|
| Parameter: Columns | propercase |
| Parameter: Formula | PROPER(input) |

### Results:

| input | uppercase | lowercase | propercase |
|---|---|---|---|
| AbCdEfGh IjKlMnO | ABCDEFGH IJKLMNO | abcdefgh ijklmno | Abcdefgh Ijklmno |
| go West, young man! | GO WEST, YOUNG MAN! | go west, young man! | Go West, Young Man! |

| | | | |
|---|---|---|---|
| Oh, *(*$%(&! That HURT! | OH, *(*$%(&! THAT HURT! | oh, *(*$%(&! that hurt! | Oh, *(*$%(&! That Hurt! |
| A11 w0rk and n0 0play makes Jack a dull boy. | A11 W0RK AND N0 0PLAY MAKES JACK A DULL BOY. | a11 w0rk and n0 0play makes jack a dull boy. | A11 W0rk And N0 0play Makes Jack A Dull Boy. |

# PROPER Function

Converts an input string to propercase. Input can be a column reference or a string literal.

**Propercase** is strict title case. If the first character of any non-breaking segment of letters in the string is an alphabetical character, it is capitalized. Otherwise, the string is unchanged. See the examples below.

**Wrangle vs. SQL:** This function is part of Wrangle , a proprietary data transformation language. Wrangle is not SQL. For more information, see *Wrangle Language*.

## Basic Usage

**Column reference example:**

```
proper(MyName)
```

**Output:** Returns the string values in the `MyName` column value written in Proper Case.

**String literal example:**

```
proper(&apos;Hello, world&apos;)
```

**Output:** The string `Hello, World` is written to the new column.

## Syntax and Arguments

```
proper(column_string)
```

| Argument | Required? | Data Type | Description |
|----------|-----------|-----------|-------------|
| column_string | Y | string | Name of the column or string literal to be applied to the function |

For more information on syntax standards, see *Language Documentation Syntax Notes*.

**column_string**

Name of the column or string constant to be converted.

- Missing string or column values generate missing string results.
- String constants must be quoted (`'Hello, World'`).
- Multiple columns and wildcards are not supported.

**Usage Notes:**

| Required? | Data Type | Example Value |
|-----------|-----------|---------------|
| Yes | String literal or column reference | `myColumn` |

## Examples

> **Tip:** For additional examples, see *Common Tasks*.

**Example - uppercase, lowercase, propercase functions**

This example demonstrates functions that can be used to change the case of String values.

**Functions:**

| Item | Description |
|------|-------------|
| UPPER Function | All alphabetical characters in the input value are converted to uppercase in the output value. |
| LOWER Function | All alphabetical characters in the input value are converted to lowercase in the output value. |
| PROPER Function | Converts an input string to propercase. Input can be a column reference or a string literal. |

**Source:**

In the following example, you can see a number of input values in the leftmost column. The output columns are blank.

| input | uppercase | lowercase | propercase |
|-------|-----------|-----------|------------|
| AbCdEfGh IjKlMnO | | | |
| go West, young man! | | | |
| Oh, *(*$%(&! That HURT! | | | |
| A11 w0rk and n0 0play makes Jack a dull boy. | | | |

**Transformation:**

To generate uppercase, lowercase, and propercase values in the output columns, use the following transforms:

| Transformation Name | Edit column with formula |
|---------------------|--------------------------|
| Parameter: Columns | uppercase |
| Parameter: Formula | UPPER(input) |

| Transformation Name | Edit column with formula |
|---------------------|--------------------------|
| Parameter: Columns | lowercase |
| Parameter: Formula | LOWER(input) |

| Transformation Name | Edit column with formula |
|---------------------|--------------------------|
| Parameter: Columns | propercase |
| Parameter: Formula | PROPER(input) |

**Results:**

| input | uppercase | lowercase | propercase |
|-------|-----------|-----------|------------|
| AbCdEfGh IjKlMnO | ABCDEFGH IJKLMNO | abcdefgh ijklmno | Abcdefgh Ijklmno |
| go West, young man! | GO WEST, YOUNG MAN! | go west, young man! | Go West, Young Man! |

| | | | |
|---|---|---|---|
| Oh, *(*$%(&! That HURT! | OH, *(*$%(&! THAT HURT! | oh, *(*$%(&! that hurt! | Oh, *(*$%(&! That Hurt! |
| A11 w0rk and n0 0play makes Jack a dull boy. | A11 W0RK AND N0 0PLAY MAKES JACK A DULL BOY. | a11 w0rk and n0 0play makes jack a dull boy. | A11 W0rk And N0 0play Makes Jack A Dull Boy. |

| | | | |
|---|---|---|---|
| Oh, *(*$%(&! That HURT! | OH, *(*$%(&! THAT HURT! | oh, *(*$%(&! that hurt! | Oh, *(*$%(&! That Hurt! |
| A11 w0rk and n0 0play makes Jack a dull boy. | A11 W0RK AND N0 0PLAY MAKES JACK A DULL BOY. | a11 w0rk and n0 0play makes jack a dull boy. | A11 W0rk And N0 0play Makes Jack A Dull Boy. |

# TRIM Function

**Contents:**

---

Removes leading and trailing whitespace from a string. Spacing between words is not removed.

- If a string begins or ends with spaces, tabs, or other non-visible characters, they are removed by this function.
- The `TRIM` function does not remove whitespace between non-whitespace values, such as spaces between words. To remove that type of whitespace, use `REMOVEWHITESPACE`. See *REMOVEWHITESPACE Function*.
- The `TRIM` function can be used with the `TRIMQUOTES` function, which removes leading and trailing single- and double-quotes. For more information, see *TRIMQUOTES Function*.

**Wrangle vs. SQL:** This function is part of Wrangle , a proprietary data transformation language. Wrangle is not SQL. For more information, see *Wrangle Language*.

## Basic Usage

**Column reference example:**

```
trim(MyName)
```

**Output:** Returns the values of the `MyName` column value with whitespace removed from the beginning and the end.

**String literal example:**

```
trim(&apos; Hello, World &apos;)
```

**Output:** Returns the string:`Hello, World`.

## Syntax and Arguments

```
trim(column_string)
```

| Argument | Required? | Data Type | Description |
|---|---|---|---|
| column_string | Y | string | Name of the column or string literal to be applied to the function |

For more information on syntax standards, see *Language Documentation Syntax Notes*.

**column_string**

Name of the column or string constant to be trimmed.

- Missing string or column values generate missing string results.
- String constants must be quoted (`'Hello, World'`).
- Multiple columns and wildcards are not supported.

**Usage Notes:**

| Required? | Data Type | Example Value |
|-----------|-----------|---------------|
| Yes | String literal or column reference | `myColumn` |

## Examples

> **Tip:** For additional examples, see *Common Tasks*.

**Example - Trimming leading and trailing whitespace**

In this example, whitespace values are identified according to this table. The ASCII value column identifies that ASCII character value that represents the character.

- The ASCII character set is a standard method for representing keyboard and special characters on the computer. For more information on ASCII, see *http://www.asciitable.com/*.

| Value | Definition | ASCII value |
|-------|-----------|-------------|
| (space) | spacebar | Char(32) |
| (tab) | tab character | Char(9) |
| (cr) | carriage return | Char(13) |
| (nl) | newline | Char(10) |

**Source:**

In the following example dataset, input values are represented in the `mystring`. The values in the table above are represented in the string values below.

| mystring |
|----------|
| Here's my string. |
| (space)(space)Here's my string.(space)(space) |
| (tab)Here's my string.(tab) |
| (cr)Here's my string.(cr) |
| (nl)Here's my string.(nl) |
| (space)(space)(tab)Here's my string.(tab)(space)(space) |
| (space)(space)(tab)(cr)Here's my string.(cr)(tab)(space)(space) |
| (space)(space)(tab)(nl)(cr)Here's my string.(cr)(nl)(tab)(space)(space) |

**Input:**

When the above CSV data is imported into the Transformer page, it is represented as the following:

| mystring |
| --- |
| Here's my string. |
| (space)(space)Here's my string.(space)(space) |
| "(tab)Here's my string.(tab)" |
| "(cr)Here's my string.(cr)" |
| "(nl)Here's my string.(nl)" |
| "(space)(space)(tab)Here's my string.(tab)(space)(space)" |
| "(space)(space)(tab)(cr)Here's my string.(cr)(tab)(space)(space)" |
| "(space)(space)(tab)(nl)(cr)Here's my string.(cr)(nl)(tab)(space)(space)" |

**Transformation:**

You might notice the quote marks around most of the imported values.

> **NOTE:** If an imported string value contains tab, carriage return, or newline values, it is bracketed by double quotes.

The first step is to remove the quote marks. You can select one of the quote marks in the data grid and then select the appropriate Replace suggestion card. The transform should look like the following:

| Transformation Name | `Replace text or pattern` |
| --- | --- |
| **Parameter: Column** | `mystring` |
| **Parameter: Find** | `` `"` `` |
| **Parameter: Replace with** | `''` |
| **Parameter: Match all occurrences** | `true` |

Now, you can apply the `trim` function:

| Transformation Name | `New formula` |
| --- | --- |
| **Parameter: Formula type** | `Single row formula` |
| **Parameter: Formula** | `trim(mystring)` |
| **Parameter: New column name** | `'trim_mystring'` |

**Results:**

In the generated `trim_mystring` column, you can see the cleaned strings:

| mystring | trim_mystring |
| --- | --- |
| Here's my string. | Here's my string. |

| | |
|---|---|
| (space)(space)Here's my string.(space)(space) | Here's my string. |
| "(tab)Here's my string.(tab)" | Here's my string. |
| "(cr)Here's my string.(cr)" | Here's my string. |
| "(nl)Here's my string.(nl)" | Here's my string. |
| "(space)(space)(tab)Here's my string.(tab)(space)(space)" | Here's my string. |
| "(space)(space)(tab)(cr)Here's my string.(cr)(tab)(space)(space)" | Here's my string. |
| "(space)(space)(tab)(nl)(cr)Here's my string.(cr)(nl)(tab)(space)(space)" | Here's my string. |

> **Tip:** If any bracketing double quotes are removed, then tab, carriage return, and newline values are trimmed by the `TRIM` function.

### Example - String cleanup functions together

This example demonstrates functions that can be used to clean up strings.

**Functions**:

| Item | Description |
|---|---|
| TRIM Function | Removes leading and trailing whitespace from a string. Spacing between words is not removed. |
| REMOVEWHITESPACE Function | Removes all whitespace from a string, including leading and trailing whitespace and all whitespace within the string. |
| REMOVESYMBOLS Function | Removes all characters from a string that are not letters, numbers, accented Latin characters, or whitespace. |

**Source:**

In the following `(space)` and `(tab)` indicate space keys and tabs, respectively. Carriage return and newline characters are also supported by whitespace functions.

| Strings | source |
|---|---|
| String01 | this source(space)(space) |
| String02 | (tab)(tab)this source |
| String03 | (tab)(tab)this source(space)(space) |
| String04 | this source's? |
| String05 | Why, you @#$%^&*()! |
| String06 | this sörce |
| String07 | (space)this sörce |
| String08 | à mañana |

**Transformation:**

The following transformation steps generate new columns using each of the string cleanup functions:

| Transformation Name | New formula |
|---|---|
| | |

| Parameter: Formula type | Single row formula |
|---|---|
| Parameter: Formula | TRIM(source) |
| Parameter: New column name | 'trim_source' |

| | |
|---|---|
| **Transformation Name** | New formula |
| **Parameter: Formula type** | Single row formula |
| **Parameter: Formula** | REMOVEWHITESPACE(source) |
| **Parameter: New column name** | 'removewhitespace_source' |

| | |
|---|---|
| **Transformation Name** | New formula |
| **Parameter: Formula type** | Single row formula |
| **Parameter: Formula** | REMOVESYMBOLS(source) |
| **Parameter: New column name** | 'removesymbols_source' |

**Results:**

| Strings | source | removesymbols_source | removewhitespace_source | trim_source |
|---|---|---|---|---|
| String01 | this source(space)(space) | this source(space)(space) | thissource | this source |
| String02 | (tab)(tab)this source | (tab)(tab)this source | thissource | this source |
| String03 | (tab)(tab)this source(space)(space) | (tab)(tab)this source(space)(space) | thissource | this source |
| String04 | this source's? | this sources | thissource's? | this source's? |
| String05 | "Why, you @#$%^&*()!" | Why you | Why,you@#$%^&*()! | Why, you @#$%^&*()! |
| String06 | this söurce | this söurce | thissöurce | this söurce |
| String07 | (space)this söurce | (space)this söurce | thissöurce | this söurce |
| String08 | à mañana | à mañana | àmañana | à  mañana |

# REMOVEWHITESPACE Function

Removes all whitespace from a string, including leading and trailing whitespace and all whitespace within the string.

Spacing between words is removed.

**Wrangle vs. SQL:** This function is part of Wrangle , a proprietary data transformation language. Wrangle is not SQL. For more information, see *Wrangle Language*.

## Basic Usage

**Column reference example:**

```
removewhitespace(MyName)
```

**Output:** Returns the value of the `MyName` column value with all whitespace removed.

**String literal example:**

```
removewhitespace(&apos; Hello, World &apos;)
```

**Output:** Returns the string: `Hello,World`.

## Syntax and Arguments

```
removewhitespace(column_string)
```

| Argument | Required? | Data Type | Description |
|---|---|---|---|
| column_string | Y | string | Name of the column or string literal to be applied to the function |

For more information on syntax standards, see *Language Documentation Syntax Notes*.

**column_string**

Name of the column or string constant to be cleaned of whitespace.

- Missing string or column values generate missing string results.
- String constants must be quoted (`'Hello, World'`).
- Multiple columns and wildcards are not supported.

**Usage Notes:**

| Required? | Data Type | Example Value |
|---|---|---|
| Yes | String literal or column reference | `myColumn` |

## Examples

> **Tip:** For additional examples, see *Common Tasks.*

### Example - String cleanup functions together

This example demonstrates functions that can be used to clean up strings.

**Functions:**

| Item | Description |
|---|---|
| TRIM Function | Removes leading and trailing whitespace from a string. Spacing between words is not removed. |
| REMOVEWHITESPACE Function | Removes all whitespace from a string, including leading and trailing whitespace and all whitespace within the string. |
| REMOVESYMBOLS Function | Removes all characters from a string that are not letters, numbers, accented Latin characters, or whitespace. |

### Source:

In the following `(space)` and `(tab)` indicate space keys and tabs, respectively. Carriage return and newline characters are also supported by whitespace functions.

| Strings | source |
|---|---|
| String01 | this source(space)(space) |
| String02 | (tab)(tab)this source |
| String03 | (tab)(tab)this source(space)(space) |
| String04 | this source's? |
| String05 | Why, you @#$%^&*()! |
| String06 | this söurce |
| String07 | (space)this söurce |
| String08 | à mañana |

### Transformation:

The following transformation steps generate new columns using each of the string cleanup functions:

| Transformation Name | New formula |
|---|---|
| **Parameter: Formula type** | Single row formula |
| **Parameter: Formula** | TRIM(source) |
| **Parameter: New column name** | 'trim_source' |

| Transformation Name | New formula |
|---|---|
| **Parameter: Formula type** | Single row formula |
| **Parameter: Formula** | REMOVEWHITESPACE(source) |

| Parameter: New column name | 'removewhitespace_source' |
|---|---|

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | REMOVESYMBOLS(source) |
| Parameter: New column name | 'removesymbols_source' |

**Results:**

| Strings | source | removesymbols_source | removewhitespace_source | trim_source |
|---|---|---|---|---|
| String01 | this source(space)(space) | this source(space)(space) | thissource | this source |
| String02 | (tab)(tab)this source | (tab)(tab)this source | thissource | this source |
| String03 | (tab)(tab)this source(space) (space) | (tab)(tab)this source(space) (space) | thissource | this source |
| String04 | this source's? | this sources | thissource's? | this source's? |
| String05 | "Why, you @#$%^&*()!" | Why you | Why,you@#$%^&*()! | Why, you @#$% ^&*()! |
| String06 | this söurce | this söurce | thissöurce | this söurce |
| String07 | (space)this söurce | (space)this söurce | thissöurce | this söurce |
| String08 | à mañana | à mañana | àmañana | à  mañana |

# REMOVESYMBOLS Function

Removes all characters from a string that are not letters, numbers, accented Latin characters, or whitespace.

> **NOTE:** Non-Latin letters are also removed.

> **Tip:** This function also removes common punctuation, such as the following:
>
> ```
> . , ! & ?
> ```
>
> To preserve these characters, you might replace them with an alphanumeric text string. For example, the question mark might be replaced by:
>
> ```
> zzQUESTIONMARKzz
> ```
>
> After the function has been applied, you can replace these strings with the original values.

**Wrangle vs. SQL:** This function is part of Wrangle , a proprietary data transformation language. Wrangle is not SQL. For more information, see *Wrangle Language*.

## Basic Usage

**Column reference example:**

```
removesymbols(MyName)
```

**Output:** Returns the value in the `MyName` column value with all non-alphanumeric characters removed.

**String literal example:**

```
removesymbols(&apos;Héllõ, Wõrlds!?!?&apos;)
```

**Output:** Returns the string: `Héllõ Wõrlds`.

**Wildcard example:**

```
removesymbols($col)
```

**Output:** Strips all non-alphanumeric or space characters from all columns in the dataset.

## Syntax and Arguments

```
removesymbols(column_string)
```

| Argument | Required? | Data Type | Description |
|---|---|---|---|
| column_string | Y | string | Name of the column or string literal to be applied to the function |

For more information on syntax standards, see *Language Documentation Syntax Notes*.

**column_string**

Name of the column or string constant to be trimmed of symbols.

- Missing string or column values generate missing string results.
- String constants must be quoted (`'Hello, World'`).
- Multiple columns and wildcards are not supported.

**Usage Notes:**

| Required? | Data Type | Example Value |
|---|---|---|
| Yes | String literal or column reference | `myColumn` |

## Examples

> **Tip:** For additional examples, see *Common Tasks*.

### Example - String cleanup functions together

This example demonstrates functions that can be used to clean up strings.

**Functions**:

| Item | Description |
|---|---|
| TRIM Function | Removes leading and trailing whitespace from a string. Spacing between words is not removed. |
| REMOVEWHITESPACE Function | Removes all whitespace from a string, including leading and trailing whitespace and all whitespace within the string. |
| REMOVESYMBOLS Function | Removes all characters from a string that are not letters, numbers, accented Latin characters, or whitespace. |

**Source:**

In the following `(space)` and `(tab)` indicate space keys and tabs, respectively. Carriage return and newline characters are also supported by whitespace functions.

| Strings | source |
|---|---|
| String01 | this source(space)(space) |
| String02 | (tab)(tab)this source |
| String03 | (tab)(tab)this source(space)(space) |
| String04 | this source's? |
| String05 | Why, you @#$%^&*()! |
| String06 | this sörce |
| String07 | (space)this sörce |
| String08 | à mañana |

**Transformation:**

The following transformation steps generate new columns using each of the string cleanup functions:

| | |
|---|---|
| **Transformation Name** | `New formula` |
| **Parameter: Formula type** | `Single row formula` |
| **Parameter: Formula** | `TRIM(source)` |
| **Parameter: New column name** | `'trim_source'` |

| | |
|---|---|
| **Transformation Name** | `New formula` |
| **Parameter: Formula type** | `Single row formula` |
| **Parameter: Formula** | `REMOVEWHITESPACE(source)` |
| **Parameter: New column name** | `'removewhitespace_source'` |

| | |
|---|---|
| **Transformation Name** | `New formula` |
| **Parameter: Formula type** | `Single row formula` |
| **Parameter: Formula** | `REMOVESYMBOLS(source)` |
| **Parameter: New column name** | `'removesymbols_source'` |

**Results:**

| Strings | source | removesymbols_source | removewhitespace_source | trim_source |
|---|---|---|---|---|
| String01 | this source(space)(space) | this source(space)(space) | thissource | this source |
| String02 | (tab)(tab)this source | (tab)(tab)this source | thissource | this source |
| String03 | (tab)(tab)this source(space)(space) | (tab)(tab)this source(space)(space) | thissource | this source |
| String04 | this source's? | this sources | thissource's? | this source's? |
| String05 | "Why, you @#$%^&*()!" | Why you | Why,you@#$%^&*()! | Why, you @#$%^&*()! |
| String06 | this söurce | this söurce | thissöurce | this söurce |
| String07 | (space)this söurce | (space)this söurce | thissöurce | this söurce |
| String08 | à mañana | à mañana | àmañana | à  mañana |

# LEN Function

Returns the number of characters in a specified string. String value can be a column reference or string literal.

**Wrangle vs. SQL:** This function is part of Wrangle , a proprietary data transformation language. Wrangle is not SQL. For more information, see *Wrangle Language*.

## Basic Usage

**Column reference example:**

```
len(MyName)
```

**Output:** Returns the number of characters in the value in column `MyName`.

**String literal example:**

```
len(&apos;Hello, World&apos;)
```

**Output:** Returns the value `12`.

## Syntax and Arguments

```
len(column_string)
```

| Argument | Required? | Data Type | Description |
|---|---|---|---|
| column_string | Y | string | Name of the column or string literal to be applied to the function |

For more information on syntax standards, see *Language Documentation Syntax Notes*.

**column_string**

Name of the column or string constant to be searched.

- Missing string or column values generate missing string results.
- String constants must be quoted (`'Hello, World'`).
- Multiple columns and wildcards are not supported.

**Usage Notes:**

| Required? | Data Type | Example Value |
|---|---|---|
| Yes | String literal or column reference | `myColumn` |

## Examples

> **Tip:** For additional examples, see *Common Tasks*.

### Example - Fixed Length Strings

**Source:**

Your product identifiers follow a specific structure that you'd like to validate in your recipe. In the following example data, the `productId` column should contain values of length `6`.

You can see that there is already a column containing validation errors for the `ProductName` column. Values in the `ProductId` column that are not this length should be flagged in a new column. Then, you should merge the two columns together to create a `ValidationError` column.

| ProductName | ProductId | ErrProductName |
|---|---|---|
| Chocolate Bunnie | 123456 | Error-ProductName |
| Chocolate Squirl | 88442286 | Error-ProductName |
| Chocolate Gopher | 12345 | |

**Transformation:**

To validate the length of the values in `ProductId`, enter the following transform. Note that the `as` parameter enables you to rename the column as part of the transform.

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | if(len(ProductId) <> 6, 'Error-length-ProductId','') |
| Parameter: New column name | 'ErrProductIdLength' |

The dataset now looks like the following:

| ProductName | ProductId | ErrProductName | ErrProductIdLength |
|---|---|---|---|
| Chocolate Bunnie | 123456 | Error-ProductName | |
| Chocolate Squirrel | 88442286 | Error-ProductName | Error-length-ProductId |
| Chocolate Gopher | 12345 | | Error-length-ProductId |

You can blend the two error columns into a single `DataValidationErrors` error column using the following `merge` transform. Note again the use of the `as` parameter:

| Transformation Name | Merge columns |
|---|---|
| Parameter: Columns | ErrProductName,ErrProductIdlength |
| Parameter: Separator | '' |
| Parameter: New column name | 'DataValidationErrors' |

To clean up the data, you might want to do the following, which trims out the whitespace in the `DataValidation Errors` column and removes the two individual error columns:

| Transformation Name | Edit column with formula |
|---|---|
| Parameter: Columns | DataValidationErrors |

| Parameter: Formula | `trim(DataValidationErrors)` |
|---|---|

| Transformation Name | `Delete columns` |
|---|---|
| Parameter: Columns | `ErrProductName,ErrProductIdLength` |
| Parameter: Action | `Delete selected columns` |

**Results:**

The final dataset should look like the following:

| ProductName | ProductId | DataValidationErrors |
|---|---|---|
| Chocolate Bunnie | 123456 | Error-ProductName |
| Chocolate Squirrel | 88442286 | Error-ProductName Error-length-ProductId |
| Chocolate Gopher | 12345 | Error-length-ProductId |

# FIND Function

**Contents:**

Returns the index value in the input string where a specified matching string is located in provided column, string literal, or function returning a string. Search is conducted left-to-right.

- A column reference can refer to a column of String, Object, or Array type, which makes the `FIND` function useful for filtering data before it has been completely un-nested into tabular data.
- Returned value is from the beginning of the string, regardless of the string index value.
- If no match is found, the function returns a null value.
- If you need to determine if a value is in an array or not, you can use the `MATCHES` function, which returns a true/false response. See *MATCHES Function*.

You can also search a string from the right. For more information, see *RIGHTFIND Function*.

**Wrangle vs. SQL:** This function is part of Wrangle , a proprietary data transformation language. Wrangle is not SQL. For more information, see *Wrangle Language*.

## Basic Usage

**Column reference example:**

```
find(MyName,'find this',true,0)
```

**Output**: Searches the `MyName` column value for the string `find this` from the beginning of the value, ignoring case. If a match is found, the index value where the string is located is returned.

**String literal example:**

```
find('Hello, World','lo',false,2)
```

**Output:** Searches the string `Hello, World` for the string `lo`, in a case-sensitive search, beginning at the third character in the string. Since the match is found at the fourth character, the value `3` is returned.

**If example:**

```
if(find(SearchPool,'FindIt') >= 0, 'found it', '')
```

**Output**: Searches the `SearchPool` column value for the string `FindIt` from the beginning of the value (default). Default behavior is to not ignore case. If the string is found, the value `found it` is returned. Otherwise, the column is empty.

## Syntax and Arguments

```
find(string_to_search, search_for,[ignore_case], [start_at])
```

| Argument | Required? | Data Type | Description |
|---|---|---|---|
| string_to_search | Y | string | Name of the column, function returning a string, or string literal to be applied to the function. |
| search_for | Y | string | The string or pattern you want to look for. This can be a string, function returning a string, or string literal or Pattern pattern or regular expression. |
| ignore_case | N | boolean | Indicates if the Find function ignores case when trying to match the string or pattern. The default value is false. |
| start_at | N | integer (non-negative) | Indicates the position in the column or string literal value at which to begin the search. This value can be an integer, a function returning an integer, or a column containing integers. The default value is 0.<br><br>If not specified, the entire string is searched. |

For more information on syntax standards, see *Language Documentation Syntax Notes*.

**string_to_search**

Name of the item to be searched. Valid values can be:

- String literals must be quoted ( 'Hello, World' ).
- Column reference to any type that can be inferred as a string, which encompasses all values.
- Function returning a string value.

Missing values generate the start-at parameter value.

- Multiple values and wildcards are not supported.

**Usage Notes:**

| Required? | Data Type | Example Value |
|---|---|---|
| Yes | String literal, function returning a string, or column reference (String, Array, or Object) | myColumn |

**search_for**

Column of strings, function returning a string, string literal or pattern to find. An input value can be a literal, Pattern , or a regular expression.

- Missing string or column values generate the start-at parameter value.
    - String literals must be quoted ('Hello, World').
- Multiple values and wildcards are not supported.
- Column names are not supported.

**Usage Notes:**

| Required? | Data Type | Example Value |
|---|---|---|
| Yes | String literal or pattern | 'Hello' |

**ignore_case**

If `true`, the FIND function ignores case when trying to match the string literal or pattern value.

Default value is `false`, which means that case-sensitive matching is performed by default.

**Usage Notes:**

| Required? | Data Type | Example Value |
|---|---|---|
| No | Boolean | `true` |

**start_at**

Indicates the position in the column or string literal value at which to begin the search. For example, a value of `2` instructs the FIND function to begin searching from the third character in the column or string value.

> **NOTE:** Index values begin at `0`. If not specified, the default value is `0`, which searches the entire string.

- Value can be an integer, a function returning an integer, or a column containing integers.
  - If a column name is specified, start_at can be different for each row.
  - If a constant integer value is specified, the start_at is same for all rows.
- Value must be a non-negative integer value.
- If this value is greater than the length of the string, then no match is possible.

**Usage Notes:**

| Required? | Data Type | Example Value |
|---|---|---|
| No | Integer (non-negative) | 2 |

## Examples

> **Tip:** For additional examples, see *Common Tasks*.

**Example - Locate product purchases in transaction stream**

**Source:**

You have the simplified transaction stream listed below in which master information about a transaction (`Transac tionId` and `CustomerId`) is paired with order detail information that is brought into the application as an array in the `OrderDetail` column. The array column contains information about product ID, quantity, and the type of transaction.

| TransactionId | CustomerId | OrderDetail |
|---|---|---|
| 12312312 | 100023 | [{"ProdId":"54321","Qty":"5","TransType":"PURCHASE"}] |
| 12312313 | 100045 | [{"ProdId":"94105","Qty":"12","TransType":"PURCHASE"}] |
| 12312314 | 100066 | [{"ProdId":"54321","Qty":"1","TransType":"TEST"}] |

| | | |
|---|---|---|
| 12312315 | 100068 | [{"ProdId":"85858","Qty":"9","TransType":"PURCHASE"}] |

The transaction stream includes test transactions, which are identified by the value `TEST` for `TransType` in the detail column. You want to remove these transactions early in the process, which should simplify your dataset and speed up its processing.

### Transformation:

First, you must identify the records that contain the test transaction value. The following transform generates a new column containing true/false values for whether the value `"TEST"` appears in the `OrderDetail` transform.

> **Tip:** You should include the double-quotes around the value, in case the other fields in the array could contain some version of the value `TEST`. Note that the double quotes need to be escaped, as in the value below.

| Transformation Name | `New formula` |
|---|---|
| Parameter: Formula type | `Single row formula` |
| Parameter: Formula | `find(OrderDetail, '\"TEST\"', false, 0)` |
| Parameter: New column name | `find_OrderDetail` |

When the step is added to the recipe, the `find_OrderDetail` column is generated, containing the index value returned by the `FIND` function. In this case, there is only one row that contains a value: `53` for the third transaction.

| TransactionId | CustomerId | OrderDetail | find_OrderDetail |
|---|---|---|---|
| 12312312 | 100023 | [{"ProdId":"54321","Qty":"5","TransType":"PURCHASE"}] | |
| 12312313 | 100045 | [{"ProdId":"94105","Qty":"12","TransType":"PURCHASE"}] | |
| 12312314 | 100066 | [{"ProdId":"54321","Qty":"1","TransType":"TEST"}] | 53 |
| 12312315 | 100068 | [{"ProdId":"85858","Qty":"9","TransType":"PURCHASE"}] | |

You can then add the following step to keep the rows where the `FIND` function returned a null value in the `find_OrderDetail` column:

| Transformation Name | `Filter rows` |
|---|---|
| Parameter: Condition | `Is missing` |
| Parameter: Column | `find_OrderDetail` |
| Parameter: Action | `Keep matching rows` |

### Results:

| TransactionId | CustomerId | OrderDetail | find_OrderDetail |
|---|---|---|---|
| 12312312 | 100023 | [{"ProdId":"54321","Qty":"5","TransType":"PURCHASE"}] | |
| 12312313 | 100045 | [{"ProdId":"94105","Qty":"12","TransType":"PURCHASE"}] | |
| 12312315 | 100068 | [{"ProdId":"85858","Qty":"9","TransType":"PURCHASE"}] | |

You can delete the `find_OrderDetail` column at this time.

# RIGHTFIND Function

**Contents:**

Returns the index value in the input string where the last instance of a matching string is located. Search is conducted right-to-left.

Input can be specified as a column reference or a string literal, although string literal usage is rare.

- A column reference can refer to a column of String, Object, or Array type, which makes the `RIGHTFIND` function useful for filtering data before it has been completely un-nested into tabular data.
- Starting location is specified from the end of the string.
- Returned value is from the beginning of the string, regardless of the string index value.
- If no match is found, the function returns a null value.
- If you need to determine if a value is in an array or not, you can use the `MATCHES` function, which returns a true/false response. See *MATCHES Function*.

You can also search a string from the left. For more information, see *FIND Function*.

**Wrangle vs. SQL:** This function is part of Wrangle , a proprietary data transformation language. Wrangle is not SQL. For more information, see *Wrangle Language*.

## Basic Usage

**Column reference example:**

```
rightfind(MyName,&apos;find this&apos;,true,0)
```

**Output**: Searches the `MyName` column value for the last instance of the string `find this` from the end of the value, ignoring case. If a match is found, the index value from the beginning of the string is returned.

**String literal example:**

```
rightfind(&apos;Hello, World&apos;,&apos;lo&apos;,false,2)
```

**Output:** Searches the string `Hello, World` for the string `lo`, in a case-sensitive search from the third-to-last character of the string. Since the match is found at the fourth character from the left, the value `3` is returned.

**If example:**

```
if(rightfind(SearchPool,&apos;FindIt&apos;) &gt;= 0, &apos;found it&apos;, &apos;&apos;)
```

**Output**: Searches the `SearchPool` column value for the string `FindIt` from the end of the value (default). Default behavior is to not ignore case. If the string is found, the value `found it` is returned. Otherwise, the value is empty.

## Syntax and Arguments

```
rightfind(string_to_search,search_for,[ignore_case], [start_at])
```

| Argument | Required? | Data Type | Description |
|---|---|---|---|
| string_to_search | Y | string | Name of the string, function returning a string, or a column containing strings. |
| search_for | Y | string | The string or pattern you want to search. This can be a string, function returning a string, or string literal or<br><br>Pattern pattern or regular expression. |
| ignore_case | N | boolean | Indicates if the `Rightfind` function ignores case when trying to match the string or pattern. The default value is `false`. |
| start_at | N | integer (non-negative) | Indicates the position in the column or string literal value at which to begin the search. This value can be an integer, a function returning an integer, or a column containing integers. The default value is 0.<br><br>If not specified, the entire string is searched. |

For more information on syntax standards, see *Language Documentation Syntax Notes*.

### string_to_search

Name of the item to be searched. Valid values can be:

- String literals must be quoted ( `'Hello, World'` ).
- column reference to any type that can be inferred as a string, which encompasses all values.

Missing values generate the start-index parameter value.

- Multiple values and wildcards are not supported.

**Usage Notes:**

| Required? | Data Type | Example Value |
|---|---|---|
| Yes | String literal or column reference (String, Array, or Object) | `myColumn` |

### search_for

String literal or pattern to find. This value can be a string literal, a Pattern , or a regular expression.

- Missing string or column values generate the start-index parameter value.
    - String literals must be quoted (`'Hello, World'`).
- Multiple values and wildcards are not supported.

**Usage Notes:**

| Required? | Data Type | Example Value |
|---|---|---|
| Yes | String literal or pattern | `'Hello'` |

**ignore_case**

If `true`, the `RIGHTFIND` function ignores case when trying to match the string literal or pattern value.

Default value is `false`, which means that case-sensitive matching is performed by default.

**Usage Notes:**

| Required? | Data Type | Example Value |
|-----------|-----------|---------------|
| No | Boolean | `true` |

**start_at**

The index of the character in the column or string literal value at which to begin the search, from the end of the string. For example, a value of `2` instructs the `RIGHTFIND` function to begin searching from the third character in the column or string value.

> **NOTE:** Index values begin at `0`. If not specified, the default value is `0`, which searches the entire string from the end of the string.

- Value can be an integer, a function returning an integer, or a column containing integers.
    - If a column name is specified, start_at can be different for each row.
    - If a constant integer value is specified, the start_at is same for all rows.
- Value must be a non-negative integer value.
- If this value is greater than the length of the string, then no match is possible.

**Usage Notes:**

| Required? | Data Type | Example Value |
|-----------|-----------------------|---------------|
| No | Integer (non-negative) | 2 |

## Examples

> **Tip:** For additional examples, see *Common Tasks*.

**Example - Locate filenames in a URL**

In this example, you must extract filenames from a column of URL values. Some rows do not have filenames, and there is some variation in the structure of the URLs.

**Source:**

| URL |
|-----|
| www.example.com |
| http://www.example.com |
| http://www.example.com/test_app |
| http://www.example.com/index.html |
| http://www.example.com/resources/mypic.jpg |

```
http://www.example.com/pages/mypage.html
http://www.example.com/resources/styles.css
www.example.com/resources/styles.css
```

**Transformation:**

To preserve the original column, you can use the following to create a working version of the source:

| Transformation Name | `New formula` |
|---|---|
| **Parameter: Formula type** | `Single row formula` |
| **Parameter: Formula** | `URL` |
| **Parameter: New column name** | `'filename'` |

You can use the following to standardize the formatting of the working column:

| Transformation Name | `Replace text or pattern` |
|---|---|
| **Parameter: Column** | `filename` |
| **Parameter: Find** | `'http:'` |
| **Parameter: Replace with** | `''` |
| **Parameter: Ignore case** | `true` |

> **Tip:** You may need to modify the above to use a Pattern to also remove `https://`.

The next two steps calculate where in the `filename` values the forward slash and dot values are located, if at all:

| Transformation Name | `New formula` |
|---|---|
| **Parameter: Formula type** | `Single row formula` |
| **Parameter: Formula** | `rightfind(filename,"\/",true,0)` |
| **Parameter: New column name** | `'rightFindSlash'` |

| Transformation Name | `New formula` |
|---|---|
| **Parameter: Formula type** | `Single row formula` |
| **Parameter: Formula** | `rightfind(filename,".",true,0)` |
| **Parameter: New column name** | `'rightFindDot'` |

If either of the above values is `0`, then there is no filename present:

| | |
|---|---|
| **Transformation Name** | `Edit column with formula` |
| **Parameter: Columns** | `filename` |
| **Parameter: Formula** | `if((rightFindSlash == 0) || (rightFindDot == 0), '', right (filename,(len(filename)-rightFindSlash)))` |

**Results:**

After removing the intermediate columns, you should end up with something like the following:

| URL | filename |
|---|---|
| www.example.com | |
| http://www.example.com | |
| http://www.example.com/test_app | |
| http://www.example.com/index.html | index.html |
| http://www.example.com/resources/mypic.jpg | mypic.jpg |
| http://www.example.com/pages/mypage.html | mypage.html |
| http://www.example.com/resources/styles.css | styles.css |
| www.example.com/resources/styles.css | styles.css |

# FINDNTH Function

**Contents:**

Returns the position of the nth occurrence of a letter or pattern in the input string where a specified matching string is located in the provided column. You can search either from left or right.

- A column reference can refer to a column of String, Object, or Array type, which makes the `FINDNTH` function useful for filtering data before it has been completely un-nested into tabular data.
- Returned value is from the beginning of the string, regardless of the string index value.
- If no match is found, the function returns a null value.

**Wrangle vs. SQL:** This function is part of Wrangle , a proprietary data transformation language. Wrangle is not SQL. For more information, see *Wrangle Language*.

## Basic Usage

**Column reference example:**

```
findnth(Message, `{hashtag}`, 2, left, true)
```

**Output**: Searches the `Message` column value for second occurrence of the `hashtag` pattern from the left of the column by considering the case-sensitive matching. If a match is found, returns the index value of the location in the string where the pattern appears.

**String literal example:**

```
findnth('Hello, World','o', 2, left ,false)
```

**Output:** Searches the string `Hello, World` for the 2nd occurrence of the value `o` from the left of the column by ignoring the case. In this case, the returned value is `8`.

## Syntax and Arguments

```
findnth(string_to_search, search_for, match_number, direction, ignore_case)
```

| Argument | Required? | Data Type | Description |
|---|---|---|---|
| string_to_search | Y | string | Name of the column, function returning a string, or string literal to be applied to the function |

| | | | |
|---|---|---|---|
| search_for | Y | string | Name of column, function returning a string, or string literal or pattern to find |
| match_number | Y | integer | Count of characters from the start of the value to include in the match |
| direction | N | string | The direction to search the string from. This can be either left or right. Default is left. |
| ignore_case | N | boolean | If `true`, a case-insensitive match is performed. Default is `false`. |

For more information on syntax standards, see *Language Documentation Syntax Notes*.

### string_to_search

Item to be searched. Valid values can be:

- String literals must be quoted ( `'Hello, World'` ).
- Column reference to any type that can be inferred as a string, which encompasses all values.
- Function returning a string value.
- Multiple values and wildcards are not supported.

**Usage Notes:**

| Required? | Data Type | Example Value |
|---|---|---|
| Yes | String literal, function returning a string, or column reference (String, Array, or Object) | `myColumn` |

### search_for

Column of strings, function returning a string, string literal or pattern to find. An input value can be a literal, Pattern , or a regular expression.

- Multiple values and wildcards are not supported.

**Usage Notes:**

| Required? | Data Type | Example Value |
|---|---|---|
| Yes | String literal or pattern | `'Hello'` |

### match_number

The number of string or pattern matches to find. For example, a value of `2` instructs the `FINDNTH` function to begin searching from the second occurrence of the character or pattern in the column or a string value.

**Usage Notes:**

| Required? | Data Type | Example Value |
|---|---|---|
| Yes | integer (non-negative) | `'2'` |

- Value must be a non-negative integer value.
- Value must a non-negative integer. If the value is `0`, then the match fails for all strings.

### direction

The direction to search the string from. The direction can be either left or right. By default, it is left.

**Usage Notes:**

| Required? | Data Type | Example Value |
|-----------|-----------|---------------|
| No | string | `left` |

### ignore_case

If `true`, the `FINDNTH` function ignores case when trying to match the string literal or pattern value.

Default value is `false`, which means that case-sensitive matching is performed by default.

**Usage Notes:**

| Required? | Data Type | Example Value |
|-----------|-----------|---------------|
| No | Boolean | `true` |

## Examples

> **Tip:** For additional examples, see *Common Tasks*.

### Example - Filter Hashtag messages

**Source:**

The table below contains your customer's tweet messages. You are interested in the second hashtag that is listed in each message.

> **Tip:** You can use either the FIND or the FINDNTH function for the first value in the string. However, you must use the FINDNTH function to find the second or later values in the string.

| User Name | Location | Messages |
|-----------|----------|----------|
| Eugenie | U.K | #dataprep  #businessintelligence #CommitToCleanData #London |
| Jeniffer | NewYork | Learn how #NewYorklife#  #bigdata  #dataprep  #NewYork # |
| Patrick | Berlin | #bigdata  #machine learning #datawrangling #Berlin |
| Christy | SanFrancisco | #predictivetransformation, #businessintelligence, #startwiththeuser, #machinelearning #SFO |
| Dave | Paris | #commitocleandata, #pivot, #aggregation, #bigdata, #dataprep, #machinelearning |

**Transformation:**

First, you must identify the records that contain the pattern `hashtag`. The following transform generates a new column containing the position of the 2nd hashtag in the Messages column. This value can be retrieved using the `{hashtag}` Trifacta pattern.

| Transformation Name | `New formula` |
|---------------------|---------------|
| **Parameter: Formula type** | `Single row formula` |

| Parameter: Formula | `FINDNTH(Message, ` `` `{hashtag}` `` `, 2, left, true)` |
|---|---|
| Parameter: New column name | `find2ndhashtag_Message` |

| User Name | Location | Messages | find2ndhashtag_Message |
|---|---|---|---|
| Eugenie | U.K | #dataprep  #businessintelligence #CommitToCleanData #London | 11 |
| Jeniffer | NewYork | Learn how #NewYorklife#  #bigdata  #dataprep  #NewYork # | 26 |
| Patrick | Berlin | #bigdata  #machinelearning #datawrangling #Berlin | 10 |
| Christy | SanFrancisco | #predictivetransformation, #businessintelligence, #startwiththeuser, #machinelearning #SFO | 28 |
| Dave | Paris | #commitocleandata, #pivot, #aggregation, #bigdata, #dataprep, #machinelearning | 20 |

When the step is added to the recipe, the `find2ndhashtag_Message` column is generated, containing the index value returned by the `FINDNTH` function. In this case, each row contains at least two instances of the `{hashtag}` Trifacta pattern, and the generated column contains the index position where it occurs in the `Message` column.

The next step is to extract the hashtag from the `Messages` column to convert into a meaningful data.

| Transformation Name | `Extract patterns` |
|---|---|
| Parameter: Column to extract from | `Messages` |
| Parameter: Option | `Custom text or pattern` |
| Parameter: Text to extract | `` `{hashtag}` `` |
| Parameter: Number of matches to extract | `2` |

> **NOTE**: The number of matches to extract parameter defaults to 1, meaning that the transformation extracts a maximum of one value from each cell. This value can be set from 1-50. In this case, you should set to the value to `2`, since you are interested in the second hashtag.

The above transformation generates two new columns with the first two extracted hashtag values from the Messages column. Next steps are:

- Delete the column containing the first hashtag value.
- Rename the `Messages2` column to be `Second_hashtag_value`.

**Results:**

| User Name | Location | Messages | Second_hashtag_value | find2ndhashtag_Message |
|---|---|---|---|---|
| Eugenie | U.K | #dataprep  #businessintelligence #CommitToCleanData #London | #businessintelligence | 11 |
| Jeniffer | NewYork | Learn how #NewYorklife#  #bigdata  #dataprep  #NewYork # | #bigdata | 26 |
| Patrick | Berlin | #bigdata  #machine #learning #datawrangling #Berlin | #machinelearning | 10 |

| Christy | SanFrancisco | #predictivetransformation, #businessintelligence, #startwiththeuser, #machinelearning #SFO | #businessintelligence | 28 |
| Dave | Paris | #commitocleandata, #pivot, #aggregation, #bigdata, #dataprep, #machinelearning | #pivot | 20 |

# SUBSTRING Function

**Contents:**

---

Matches some or all of a string, based on the user-defined starting and ending index values within the string.

- Input must be a string literal value.
- Since the `SUBSTRING` function matches based on fixed numeric values, changes to the length or structure of a data field can cause your recipe to fail to properly execute.
- The `SUBSTRING` function requires numerical values for the starting and ending values. If you need to match strings using patterns, you should use the `extract` transform instead. See *Extract Transform*.

**Wrangle vs. SQL:** This function is part of Wrangle , a proprietary data transformation language. Wrangle is not SQL. For more information, see *Wrangle Language*.

## Basic Usage

```
substring('Hello, World',0,5)
```

**Output:** Returns the string: `Hello`.

## Syntax and Arguments

```
substring(string_val,start_index,end_index)
```

| Argument | Required? | Data Type | Description |
|----------|-----------|-----------|-------------|
| string_val | Y | string | String literal to be applied to the function |
| start_index | Y | integer (non-negative) | Index value for the start character from the source column or value |
| end_index | Y | integer (non-negative) | Index value for the end character from the source column or value |

For more information on syntax standards, see *Language Documentation Syntax Notes*.

**string_val**

String constant to be searched.

- Missing string values generate missing string results.
- String constants must be quoted (`'Hello, World'`).
- Multiple columns and wildcards are not supported.

**Usage Notes:**

| Data Type | Required? | Example Value |
|---|---|---|
| String | Yes | `'This is my string.'` |

**start_index**

Index value of the character in the string to begin the substring match.

- The index of the first character of the string is `0`.
- Value must be less than `end_index`.
- If this value is greater than the length of the string, a missing value is returned.

**Usage Notes:**

| Data Type | Required? | Example Value |
|---|---|---|
| Integer (non-negative) | Yes | 0 |

**end_index**

Index value of the character in the string that is one after the end the substring match.

- Value must be greater than `start_index`.
- If this value is greater than the length of the string, the end of the string is the end of match. If you know the maximum length of your data, you can use that value here.

**Usage Notes:**

| Data Type | Required? | Example Value |
|---|---|---|
| Integer (non-negative) | Yes | 5 |

## Examples

> **Tip:** For additional examples, see *Common Tasks*.

**Example - Sectional Information in Zipcodes**

**Source:**

A US zip code contains five digits with an optional Zip+4 extension consisting of four digits. Valid zip code values can be a mixture of these formats.

Within zip code values, each digit has significance:

- Digit 1: Zip code section
- Digits 2-3: Region within section
- Digits 4-5: area or town within region
- Digits 6-9: Optional Zip+4 identifier within area or town

Here is some example data:

| LastName | ZipCode |
|----------|---------|
| Able | 94101 |
| Baker | 23502-1122 |
| Charlie | 36845 |

**Transformation:**

You are interested in the region and area or town identifiers within a zip code region. You can use the following transformations applied to the `ZipCode` column to extract this information:

| Transformation Name | New formula |
|---------------------|-------------|
| **Parameter: Formula type** | Single row formula |
| **Parameter: Formula** | substring(ZipCode,1,3) |

| Transformation Name | New formula |
|---------------------|-------------|
| **Parameter: Formula type** | Single row formula |
| **Parameter: Formula** | substring(ZipCode,3,5) |

Since the string can be five or ten characters in length, you need to use the `SUBSTRING` function in the second transformation, too. If the data is limited to five-digit zip codes, you could use the `RIGHT` function.

**Results:**

| LastName | ZipCode | substring_ZipCode | substring_ZipCode2 |
|----------|---------|-------------------|--------------------|
| Able | 94101 | 41 | 01 |
| Baker | 23502-1122 | 35 | 02 |
| Charlie | 36845 | 68 | 45 |

# SUBSTITUTE Function

**Contents:**

---

Replaces found string literal or pattern or column with a string, column, or function returning strings.

Input can be specified as a column reference, a function returning a string, or a string literal, although string literal usage is rare.

- A column reference can refer to a column of String type.
- If no match is found, the function returns the source string.
- If multiple matches are found in a single string, all replacements are made.

**Wrangle vs. SQL:** This function is part of Wrangle , a proprietary data transformation language. Wrangle is not SQL. For more information, see *Wrangle Language.*

## Basic Usage

**Column reference example:**

```
substitute(myURL,`{ip-address}`,myDomain)
```

**Output**: Searches the `myURL` column values for sub-strings that match valid IP addresses. Where matches are found, they are replaced with the corresponding value in the `myDomain` column.

**Function reference example:**

```
substitute(upper(companyName),&apos;ACME&apos;,&apos;New ACME&apos;)
```

**Output:** Searches the uppercase version of values from the `companyName` column for the string literal `ACME`. When found, these matches are replaced by `New ACME` in the `companyName` column.

## Syntax and Arguments

```
substitute(string_source,string_pattern,string_replacement[,ignore_case, pattern_before,
pattern_after])
```

| Argument | Required? | Data Type | Description |
|----------|-----------|-----------|-------------|
|          |           |           |             |

| string_source | Y | string | Name of the column, a function returning a string, or string literal to be applied to the function |
|---|---|---|---|
| string_pattern | Y | string | String literal or pattern or a column or a function returning strings to find |
| string_replacement | Y | string | String literal, column or function returning a string to use as replacement |
| ignore_case | N | string | When `true`, matching is case-insensitive. Default is `false`. |
| pattern_before | N | string | String literal or pattern to find before finding the string_pattern value. |
| pattern_after | N | string | String literal or pattern to find after finding the string_pattern value. |

For more information on syntax standards, see *Language Documentation Syntax Notes*.

### string_source

Name of the item to be searched. Valid values can be:

- String literals must be quoted ( `'Hello, World'` ).
- Column reference to any type that can be inferred as a string, which encompasses all values
- Functions that return string values

Multiple values and wildcards are not supported.

**Usage Notes:**

| Required? | Data Type | Example Value |
|---|---|---|
| Yes | String literal or column reference (String, Array, or Object) | `myColumn` |

### string_pattern

String to find. This value can be a string literal, a  Pattern , a regular expression, a column, or a function returning a String value.

- String literals must be quoted (`'Hello, World'`).
- Multiple values and wildcards are not supported.

**Usage Notes:**

| Required? | Data Type | Example Value |
|---|---|---|
| Yes | String value or pattern or column reference (String) | `'Hello'` |

### string_replacement

Value with which to replacement any matched patterns. Value can be a string, a function returning string values, or a column reference containing strings.

- String literals must be quoted (`'Hello, World'`).
- Column reference to any type that can be inferred as a string, which encompasses all values.
- Multiple values and wildcards are not supported.

**Usage Notes:**

| Required? | Data Type | Example Value |
|---|---|---|
| Yes | String literal, column reference (String, Array, or Object), or function returning String value | `'##REDACTED##'` |

### ignore_case

When `true`, matches are case-insensitive. Default is `false`.

> **NOTE:** This argument is not required. By default, matches are case-sensitive.

**Usage Notes:**

| Required? | Data Type | Example Value |
|---|---|---|
| No | String value | `'false'` |

### pattern_before

String literal or pattern to find in a position before the pattern to match.

> **NOTE:** This argument is not permitted when `string_ pattern` or `string_replacement` is of column data type.

> **Tip:** Use this argument if there are potentially multiple instances of the pattern to match in the source.

**Usage Notes:**

| Required? | Data Type | Example Value |
|---|---|---|
| No | String literal or pattern | `` `{digit}{3}` `` |

### pattern_after

String literal or pattern to find in a position after the pattern to match.

> **NOTE:** This argument is not permitted when `string_ pattern` or `string_replacement` is of column data type.

> **Tip:** Use this argument if there are potentially multiple instances of the pattern to match in the source.

**Usage Notes:**

| Required? | Data Type | Example Value |
|---|---|---|

| No | String literal or pattern | ' ' |
|---|---|---|

## Examples

> **Tip:** For additional examples, see *Common Tasks*.

### Example - Partial obfuscation of credit card numbers

**Source:**

Suppose you have the following transactional data, which contains customer credit card numbers.

| TransactionId | CreditCardNum | AmtDollars |
|---|---|---|
| T001 | 4111-1111-1111-1111 | 100.29 |
| T002 | 5500-0000-0000-0004 | 510.21 |
| T003 | 3400-0000-0000-009 | 162.13 |
| T004 | 3000-0000-0000-04 | 294.12 |

For security purposes, you wish to redact the first three sets of digits, so only the last set of digits appears.

**Transformation:**

To make the substitution, you must first change the type of the column to be a string:

| Transformation Name | Change column data type |
|---|---|
| Parameter: Columns | CreditCardNum |
| Parameter: New type | 'String' |

You can then use the following transformation to perform the pattern-based replacement of four-digit sets that end in a dash with XXXX:

| Transformation Name | Edit column with formula |
|---|---|
| Parameter: Columns | CreditCardNum |
| Parameter: Formula | substitute(CreditCardNum, `{digit}+\-`, 'XXXX-') |

To indicate that the column no longer contains valid information, you might choose to rename it like in the following:

| Transformation Name | Rename columns |
|---|---|
| Parameter: Option | Manual rename |
| Parameter: Column | CreditCardNum |
| Parameter: New column name | 'CreditCardNumOBSCURED' |

**Results:**

| TransactionId | CreditCardNumOBSCURED | AmtDollars |
| --- | --- | --- |
| T001 | XXXX-XXXX-XXXX-1111 | 100.29 |
| T002 | XXXX-XXXX-XXXX-0004 | 510.21 |
| T003 | XXXX-XXXX-XXXX-009 | 162.13 |
| T004 | XXXX-XXXX-XXXX-04 | 294.12 |

# LEFT Function

**Contents:**

- *Basic Usage*
- *Syntax and Arguments*
  - *column_string*
  - *char_count*
- *Examples*
  - *Example - Driver's License Type*

---

Matches the leftmost set of characters in a string, as specified by parameter. The string can be specified as a column reference or a string literal.

- Since the `LEFT` function matches based on fixed numeric values, changes to the length or structure of a data field can cause your recipe to fail to properly execute.
- The `LEFT` function requires an integer value for the number of characters to match. If you need to match strings using patterns, you should use the `STARTSWITH` transform instead. See *STARTSWITH Function*.

**Wrangle vs. SQL:** This function is part of Wrangle , a proprietary data transformation language. Wrangle is not SQL. For more information, see *Wrangle Language*.

## Basic Usage

**Column reference example:**

```
left(MyName,3)
```

**Output:** Returns the first three letters of the `MyName` column value.

**String literal example:**

```
left(&apos;Hello, World&apos;,5)
```

**Output:** Returns the string: `Hello`.

## Syntax and Arguments

```
left(column_string,char_count)
```

| Argument | Required? | Data Type | Description |
|---|---|---|---|
| column_string | Y | string | Name of the column or string literal to be applied to the function |
| char_count | Y | integer (positive) | Count of characters from the start of the value to include in the match |

For more information on syntax standards, see *Language Documentation Syntax Notes*.

**column_string**

Name of the column or string constant to be searched.

- Missing string or column values generate missing string results.
- String constants must be quoted (`'Hello, World'`).
- Multiple columns and wildcards are not supported.

**Usage Notes:**

| Required? | Data Type | Example Value |
|---|---|---|
| Yes | String literal or column reference | `myColumn` |

**char_count**

Count of characters from the start of the string to include in the match.

- Value must a non-negative integer. If the value is `0`, then the match fails for all strings.
- If this value is greater than the length of the string, then the match is the entire string.
- References to columns of integer data type are not supported.

**Usage Notes:**

| Required? | Data Type | Example Value |
|---|---|---|
| Yes | Integer (non-negative) | 5 |

## Examples

> **Tip:** For additional examples, see *Common Tasks*.

**Example - Driver's License Type**

**Source:**

A California driver license number is one alphabetical character followed by seven digits (e.g., A1234567). The following is a set of California driver's license values:

| LastName | LicenseID |
|---|---|
| Able | A1234567 |
| Baker | B5555555 |
| Charlie | C0123456 |

The `LicenseID` value contains the license class as the first character of the value. For example, Baker's license is a Commercial Class B license.

**Transformation:**

To extract the license type into a separate column, you can use the following transformation:

| Transformation Name | `New formula` |
|---|---|
| Parameter: Formula type | `Single row formula` |

| Parameter: Formula | `left(LicenseID,1)` |
|---|---|

**Results:**

| LastName | LicenseID | left_LicenseID |
|---|---|---|
| Able | A1234567 | A |
| Baker | B5555555 | B |
| Charlie | C0123456 | C |

You can rename the new column to `LicenseType`.

# RIGHT Function

**Contents:**

---

Matches the right set of characters in a string, as specified by parameter. The string can be specified as a column reference or a string literal.

- Since the `RIGHT` function matches based on fixed numeric values, changes to the length or structure of a data field can cause your recipe to fail to properly execute.
- The `RIGHT` function requires an integer value for the number of characters to match. If you need to match strings using patterns, you should use the `ENDSWITH` transform instead. See *ENDSWITH Function*.

**Wrangle vs. SQL:** This function is part of Wrangle , a proprietary data transformation language. Wrangle is not SQL. For more information, see *Wrangle Language*.

## Basic Usage

**Column reference example:**

```
right(MyString,3)
```

**Output:** Returns the rightmost (last) three letters of the `MyName` column value.

**String literal example:**

```
right(&apos;Hello, World&apos;,5)
```

**Output:** Returns the string: `World`.

## Syntax and Arguments

```
right(column_string,end_count)
```

| Argument | Required? | Data Type | Description |
|---|---|---|---|
| column_string | Y | string | Name of the column or string literal to be applied to the function |
| end_count | Y | integer (positive) | Count of characters from the end of the source string to apply to the match |

For more information on syntax standards, see *Language Documentation Syntax Notes*.

**column_string**

Name of the column or string constant to be searched.

- Missing string or column values generate missing string results.
- String constants must be quoted (`'Hello, World'`).
- Multiple columns and wildcards are not supported.

**Usage Notes:**

| Required? | Data Type | Example Value |
|---|---|---|
| Yes | String literal or column reference | `myColumn` |

### end_count

Count of characters from the right end of the string to include in the match.

- Value must a non-negative integer. If the value is `0`, then the match fails for all strings.
- If this value is greater than the length of the string, then the match is the entire string.
- References to columns of integer data type are not supported.

**Usage Notes:**

| Required? | Data Type | Example Value |
|---|---|---|
| Yes | Integer (non-negative) | 5 |

## Examples

> **Tip:** For additional examples, see *Common Tasks.*

### Example - Parse segments of social security numbers

Social security numbers follow a regular format:

```
XXX-XX-XXXX
```

Each of the separate numeric groups corresponds to a specific meaning:

- XXX - Area value that corresponds to a geographic location that surrounds the SSN applicant's address
- XX - Group number identifies the order in which the numbers are assigned within an area
- XXX - Serial number of the individual within the area and group groupings.
- For more information, see *http://www.usrecordsearch.com/ssn.htm.*

**Source:**

You want to analyze some social security numbers for area, group, and serial information. However, your social security number data is messy:

> **NOTE:** The following sample contains invalid social security numbers for privacy reasons. If you use this data in the application, it fails validation for the SSN data type.

| ParticipantId | SocialNum |
|---|---|
| 1001 | 805-88-2013 |
| 1002 | 845221914 |

| | |
|---|---|
| 1003 | 865 22 9291 |
| 1004 | 892-732213 |

**Transformation:**

When the above data is imported, the `SocialNum` column might or might not be inferred as SSN data type. Either way, you should clean up your data, using the following transforms:

| Transformation Name | `Replace text or pattern` |
|---|---|
| Parameter: **Column** | `SocialNum` |
| Parameter: **Find** | `'-'` |
| Parameter: **Replace with** | `''` |
| Parameter: **Match all occurrences** | `true` |

| Transformation Name | `Replace text or pattern` |
|---|---|
| Parameter: **Column** | `SocialNum` |
| Parameter: **Find** | `' '` |
| Parameter: **Replace with** | `''` |
| Parameter: **Match all occurrences** | `true` |

At this point, your `SocialNum` data should be inferred as SSN type and consistently formatted as a set of digits:

| ParticipantId | SocialNum |
|---|---|
| 1001 | 805882013 |
| 1002 | 845221914 |
| 1003 | 865229291 |
| 1004 | 892732213 |

From this more consistent data, you can now break out the area, group, and serial values from the column:

| Transformation Name | `New formula` |
|---|---|
| Parameter: **Formula type** | `Single row formula` |
| Parameter: **Formula** | `left(SocialNum, 3)` |
| Parameter: **New column name** | `'SSN_area'` |

| Transformation Name | `New formula` |
|---|---|

| Parameter: Formula type | `Single row formula` |
|---|---|
| Parameter: Formula | `substring(SocialNum, 3,5)` |
| Parameter: New column name | `'SSN_group'` |

| Transformation Name | `New formula` |
|---|---|
| Parameter: Formula type | `Single row formula` |
| Parameter: Formula | `right(SocialNum, 4)` |
| Parameter: New column name | `'SSN_serial'` |

If desired, you can re-order the three new columns and delete the source column:

| Transformation Name | `Move columns` |
|---|---|
| Parameter: Column(s) | `SSN_serial` |
| Parameter: Option | `After` |
| Parameter: Column | `SSN_area` |

| Transformation Name | `Move columns` |
|---|---|
| Parameter: Column(s) | `SSN_group` |
| Parameter: Option | `After` |
| Parameter: Column | `SSN_area` |

| Transformation Name | `Delete columns` |
|---|---|
| Parameter: Columns | `SocialNum` |
| Parameter: Action | `Delete selected columns` |

**Results:**

If you complete the previous transform steps, your data should look like the following:

| ParticipantId | SSN_area | SSN_group | SSN_serial |
|---|---|---|---|
| 1001 | 805 | 88 | 2013 |
| 1002 | 845 | 22 | 1914 |
| 1003 | 865 | 22 | 9291 |
| 1004 | 892 | 73 | 2213 |

# PAD Function

**Contents:**

---

Pads string values to be a specified minimum length by adding a designated character to the left or right end of the string. Returned value is of String type.

If an input value is longer than the minimum length, no change is made to the string. If you need to fit the string to be a specific length, you can use the LEFT, RIGHT, or SUBSTRING functions.

> **Tip:** You can apply the following strings after you have applied padding to ensure all values are of the same length.

- See *LEFT Function*.
- See *RIGHT Function*.
- See *SUBSTRING Function*.

**Wrangle vs. SQL:** This function is part of Wrangle , a proprietary data transformation language. Wrangle is not SQL. For more information, see *Wrangle Language*.

## Basic Usage

**Column reference example:**

```
pad(Whse_Nbr, 6, &apos;0&apos;, left)
```

**Output:** Returns a value of a minimum of six characters in length. For input values that are shorter, the character `0` is added to the left side of the string.

**String literal example:**

```
pad(&apos;My Name&apos;, 10, &apos;!&apos;, right)
```

**Output:** Returns the string: `My Name!!!!`.

## Syntax and Arguments

```
pad(string_val,string_length,pad_string,pad_side)
```

| Argument | Required? | Data Type | Description |
|---|---|---|---|

| string_val | Y | string | Name of the column, function returning string values, or string literal to be applied to the function |
|---|---|---|---|
| string_length | Y | integer (positive) | Minimum number of characters in the output string. |
| pad_string | N | string | String, column reference, or function returning a string to apply to strings that are less than the minimum length. Default is whitespace. |
| pad_side | N | enum | <ul><li>`left` - any padding is applied to the left side of the string (default)</li><li>`right` - any padding is applied to the right side of the string</li></ul> |

For more information on syntax standards, see *Language Documentation Syntax Notes*.

### string_val

Name of the column, function returning a string, or string constant to be padded.

- Missing string or column values generate missing string results.
- String constants must be quoted (`'Hello, World'`).
- Multiple columns and wildcards are not supported.

#### Usage Notes:

| Required? | Data Type | Example Value |
|---|---|---|
| Yes | String literal or column reference | `myColumn` |

### string_length

Minimum length of the generated string. Value is padded to this length at a minimum.

> **NOTE:** For input string values that are longer than the minimum string length, no padding is applied.

- Negative values have no effect on the input string.
- References to columns of integer data type are not supported.

#### Usage Notes:

| Required? | Data Type | Example Value |
|---|---|---|
| Yes | Integer (non-negative) | 5 |

### pad_string

The string of one or more characters that are used to pad input strings. If no value is provided, the default pad string is a single whitespace character.

Input values can be a string literal, a function returning a string, or a column containing strings.

#### Multi-character pad string behaviors:

When the pad string contains multiple characters, the behaviors are different depending on the side on which the string is padded:

| Function | Output Value |
|---|---|

| | |
|---|---|
| `pad('12', 4, 'abc', left)` | bc12 |
| `pad('12', 4, 'abc', right)` | 12ab |
| `pad('12', 6, 'abc', left)` | cabc12 |
| `pad('12', 6, 'abc', right)` | 12abca |

**Usage Notes:**

| Required? | Data Type | Example Value |
|---|---|---|
| Yes | String literal, function returning a string, or column reference | `'X'` |

### pad_side

An enumerated value used to determine the side of the string to which any padding is applied:

| Value | Description |
|---|---|
| `left` | Any padding is applied to the left side. This is the default value if not specified. |
| `right` | Any padding is applied to the right side. |

**Usage Notes:**

| Required? | Data Type | Example Value |
|---|---|---|
| No | One of the following: `left` or `right` | `left` |

## Examples

> **Tip:** For additional examples, see *Common Tasks*.

### Example - Numeric identifiers

In the following example, a table containing four-character product identifiers and product names has been imported into Designer Cloud Powered by Trifacta® Enterprise Edition. Unfortunately, these product identifiers are numeric in structure and are therefore interpreted by Designer Cloud Powered by Trifacta Enterprise Edition a s integer values during import. The leading zeroes are dropped for some of the values, while the latter rows in the table contain fully defined numeric values.

**Source:**

| prodId | prodName |
|---|---|

| | |
|---|---|
| 1 | Our First Product |
| 2 | Our Second Product |
| 3 | Our First Product v2 |
| 1001 | A New Product Line |
| 1002 | A New Product Line v2 |

**Transformation:**

The first step is to convert the product identifiers to string values:

| | |
|---|---|
| **Transformation Name** | `Change column data type` |
| **Parameter: Columns** | `prodId` |
| **Parameter: New type** | `'String'` |

Then, you can apply the character `0` as padding to the left of these strings, so that all values are four characters in length at a minimum:

| | |
|---|---|
| **Transformation Name** | `Edit column with formula` |
| **Parameter: Columns** | `ProdId` |
| **Parameter: Formula** | `pad(prodId,4,'0',left)` |

**Results:**

| prodId | prodName |
|---|---|
| 0001 | Our First Product |
| 0002 | Our Second Product |
| 0003 | Our First Product v2 |
| 1001 | A New Product Line |
| 1002 | A New Product Line v2 |

# MERGE Function

**Contents:**

Merges two or more columns of String type to generate output of String type. Optionally, you can insert a delimiter between the merged values.

> **NOTE:** This function behaves exactly like the `merge` transform, although the syntax is different. See *Merge Transform*.

**Wrangle vs. SQL:** This function is part of Wrangle , a proprietary data transformation language. Wrangle is not SQL. For more information, see *Wrangle Language*.

## Basic Usage

**String literal reference example:**

```
merge(['Hello,','World'],' ')
```

**Output:** Returnsthe value `Hello, World`.

**Column reference example:**

```
merge([string1,string2])
```

**Output:** Returns a single String value that is the merge of `string1` and `string2` values.

## Syntax and Arguments

```
merge([string_ref1,string_ref2],'string_delim')
```

| Argument | Required? | Data Type | Description |
|---|---|---|---|
| string_ref1 | Y | string | Name of first column or first string literal to apply to the function |
| string_ref2 | Y | string | Name of second column or second string literal to apply to the function |
| string_delim | N | string | Optional delimiter string to insert between column or literal values |

For more information on syntax standards, see *Language Documentation Syntax Notes*.

**string_ref1, string_ref2**

String literal or name of the string column whose elements you want to merge together. You can merge together two or more strings.

**Usage Notes:**

| Required? | Data Type | Example Value |
|---|---|---|
| Yes | String literal or column reference | `myString1, myString2` |

**string_delim**

Optional string literal to insert between each string that is being merged.

**Usage Notes:**

| Required? | Data Type | Example Value |
|---|---|---|
| No | String literal | `'-'` |

## Examples

> **Tip:** For additional examples, see *Common Tasks*.

**Example - Simple merge example**

The following example contains the names of a set of American authors. You need to bring together these column values into a new column, called `FullName`.

**Source:**

| FirstName | LastName | MiddleInitial |
|---|---|---|
| Jack | Kerouac | L |
| Paul | Theroux | E |
| J.D. | Salinger | |
| Philip | Dick | K |

**Transformation:**

| Transformation Name | `New formula` |
|---|---|
| Parameter: Formula type | `Single row formula` |
| Parameter: Formula | `merge([FirstName,MiddleInitial,LastName],' ')` |
| Parameter: New column name | `'FullName'` |

Since the entry for J.D. Salinger has no middle name, you might want to add the following transformation:

| Transformation Name | Replace text or pattern |
|---|---|
| Parameter: Column | FullName |
| Parameter: Find | ' ' |
| Parameter: Replace with | '' |

**Results:**

| FirstName | LastName | MiddleInitial | FullName |
|---|---|---|---|
| Jack | Kerouac | L | Jack L Kerouac |
| Paul | Theroux | E | Paul E Theroux |
| J.D. | Salinger | | J.D. Salinger |
| Philip | Dick | K | Philip K DIck |

### Other Examples

While the syntax may be different, the `MERGE` function behaves exactly like the `merge` transform. For more examples, see *Merge Transform*.

# STARTSWITH Function

**Contents:**

- *Basic Usage*
- *Syntax and Arguments*
    - *column_any*
    - *pattern*
    - *ignore_case*
- *Examples*
    - *Example - STARTSWITH and ENDSWITH Functions*

---

Returns `true` if the leftmost set of characters of a column of values matches a pattern. The source value can be any data type, and the pattern can be a Pattern , regular expression, or a string.

- The `STARTSWITH` function is ideal for matching based on patterns for any data type. If you need to match strings using a fixed number of characters, you should use the `LEFT` function instead. See *LEFT Function*.
- See *ENDSWITH Function*.

**Wrangle vs. SQL:** This function is part of Wrangle , a proprietary data transformation language. Wrangle is not SQL. For more information, see *Wrangle Language*.

## Basic Usage

**String literal example:**

```
startswith(FullName,&apos;Mr.&apos;)
```

**Output:** Returns `true` if the first three letters of the `FullName` column value are "Mr.".

**Pattern  example:**

```
startswith(CustId,`{alpha-numeric}{6}`)
```

**Output:** Returns `true` if the `CustId` column begins with a six-digit alpha-numeric sequence. Otherwise, the value is set to `false`.

**Regular expression example:**

```
if(startswith(phone,/^(\+0?1\s)?\(?\d{3}\)?[\s.-]\d{3}[\s.-]\d{4}$/),&apos;phone -
ok&apos;,&apos;phone - error&apos;)
```

**Output:** Returns `phone - ok` if the value of the `phone` column begins with a value that matches a 10-digit U.S. phone number. Otherwise, the output value is set to `phone - error`.

## Syntax and Arguments

```
startswith(column_any,pattern[,ignore_case])
```

| Argument | Required? | Data Type | Description |
|----------|-----------|-----------|-------------|

| | | | |
|---|---|---|---|
| column_any | Y | any | Name of the column to be applied to the function |
| pattern | Y | string | Pattern or literal expressed as a string describing the pattern to which to match. |
| ignore_case | N | string | When `true`, matching is case-insensitive. Default is `false`. |

For more information on syntax standards, see *Language Documentation Syntax Notes*.

### column_any

Name of the column to be searched.

- Multiple columns and wildcards are not supported.

**Usage Notes:**

| Required? | Data Type | Example Value |
|---|---|---|
| Yes | Column reference | `myColumn` |

### pattern

Pattern , regular expression, or string literal to locate in the values in the specified column.

**Usage Notes:**

| Required? | Data Type | Example Value |
|---|---|---|
| Yes | String | `` `{zip}` `` |

### ignore_case

When `true`, matches are case-insensitive. Default is `false`.

> **NOTE:** This argument is not required. By default, matches are case-sensitive.

**Usage Notes:**

| Required? | Data Type | Example Value |
|---|---|---|
| No | String value | `'false'` |

## Examples

> **Tip:** For additional examples, see *Common Tasks*.

### Example - STARTSWITH and ENDSWITH Functions

This example demonstrates functions that can be used to evaluate the beginning and end of values of any type using patterns.

**Functions**:

| Item | Description |
|------|-------------|
| STARTSWIT H Function | Returns `true` if the leftmost set of characters of a column of values matches a pattern. The source value can be any data type, and the pattern can be a Pattern , regular expression, or a string. |
| ENDSWITH Function | Returns `true` if the rightmost set of characters of a column of values matches a pattern. The source value can be any data type, and the pattern can be a Pattern , regular expression, or a string. |

**Source:**

The following inventory report indicates available quantities of product by product name. You need to verify that the product names are valid according to the following rules:

- A product name must begin with a three-digit numeric brand identifier, followed by a dash.
- A product name must end with a dash, followed by a six-digit numeric SKU.

Source data looks like the following, with the Validation column having no values in it.

| InvDate | ProductName | Qty | Validation |
|---------|-------------|-----|------------|
| 04/21/2017 | 412-Widgets-012345 | 23 | |
| 04/21/2017 | 04-Fidgets-120341 | 66 | |
| 04/21/2017 | 204-Midgets-4421 | 31 | |
| 04/21/2017 | 593-Gidgets-402012 | 24 | |

**Transformation:**

In this case, you must evaluate the `ProductName` column for two conditions. These conditional functions are the following:

```
IF(STARTSWITH(ProductName, `#{3}-`), 'Ok', 'Bad ProductName-Brand')
```

```
IF(ENDSWITH(ProductName, `-#{6}`), 'Ok', 'Bad ProductName-SKU')
```

One approach is  to create two new test columns and then edit the column based on the evaluation of these two columns. However, using the following, you can compress the evaluation into a single step without creating the intermediate columns:

| Transformation Name | Edit column with formula |
|---------------------|--------------------------|
| Parameter: Columns | Status |
| Parameter: Formula | IF(STARTSWITH(ProductName, `#{3}-`), IF(ENDSWITH(ProductName, `-#{6}`), 'Ok', 'Bad ProductName-SKU'), 'Bad ProductName-Brand') |

**Results:**

| InvDate | ProductName | Qty | Validation |
|---------|-------------|-----|------------|

| | | | |
|---|---|---|---|
| 04/21/2017 | 412-Widgets-012345 | 23 | Ok |
| 04/21/2017 | 04-Fidgets-120341 | 66 | Bad ProductName-Brand |
| 04/21/2017 | 204-Midgets-4421 | 31 | Bad ProductName-SKU |
| 04/21/2017 | 593-Gidgets-402012 | 24 | Ok |

# ENDSWITH Function

**Contents:**

---

Returns `true` if the rightmost set of characters of a column of values matches a pattern. The source value can be any data type, and the pattern can be a Pattern , regular expression, or a string.

- The `ENDSWITH` function is ideal for matching based on patterns for any data type. If you need to match strings using a fixed number of characters, you should use the `RIGHT` function instead. See *RIGHT Function*.
- See *STARTSWITH Function*.

**Wrangle vs. SQL:** This function is part of Wrangle , a proprietary data transformation language. Wrangle is not SQL. For more information, see *Wrangle Language*.

## Basic Usage

**String literal example:**

```
endswith(tweets,&apos;?&apos;)
```

**Output:** Returns `true` if last letter of the `tweets` column value is "?".

**Pattern  example:**

```
endswith(tweets,`{hashtag}{1,9}`)
```

**Output:** Returns `true` if the `tweets` column ends with 1-9 hashtag values. Otherwise, the returned value is `false` .

**Regular expression example:**

```
if(endswith,myNum,/([01][0-9][0-9]|2[0-4][0-9]|25[0-5])/),&apos;myNum - valid&apos;,&apos;
myNum - error&apos;)
```

**Output:** Returns `myNum - valid` if the value of the `myNum` column ends with a value between 0-255. Otherwise, `myNum - error` is returned.

## Syntax and Arguments

```
endswith(column_any,pattern<span>[,ignore_case]</span>)
```

| Argument | Required? | Data Type | Description |
|----------|-----------|-----------|-------------|
| column_any | Y | any | Name of the column to be applied to the function |
| pattern | Y | string | Pattern or literal expressed as a string describing the pattern to which to match. |
| ignore_case | N | string | When `true`, matching is case-insensitive. Default is `false`. |

For more information on syntax standards, see *Language Documentation Syntax Notes*.

### column_any

Name of the column to be searched.

- Multiple columns and wildcards are not supported.

### Usage Notes:

| Required? | Data Type | Example Value |
|-----------|-----------|---------------|
| Yes | Column reference | `myColumn` |

### pattern

Trifacta pattern, regular expression, or string literal to locate in the values in the specified column.

### Usage Notes:

| Required? | Data Type | Example Value |
|-----------|-----------|---------------|
| Yes | String | `` `{zip}` `` |

### ignore_case

When `true`, matches are case-insensitive. Default is `false`.

> **NOTE:** This argument is not required. By default, matches are case-sensitive.

### Usage Notes:

| Required? | Data Type | Example Value |
|-----------|-----------|---------------|
| No | String value | `'false'` |

## Examples

> **Tip:** For additional examples, see *Common Tasks*.

### Example - STARTSWITH and ENDSWITH Functions

This example demonstrates functions that can be used to evaluate the beginning and end of values of any type using patterns.

**Functions**:

| Item | Description |
|------|-------------|
| STARTSWITH Function | Returns `true` if the leftmost set of characters of a column of values matches a pattern. The source value can be any data type, and the pattern can be a Pattern , regular expression, or a string. |
| ENDSWITH Function | Returns `true` if the rightmost set of characters of a column of values matches a pattern. The source value can be any data type, and the pattern can be a Pattern , regular expression, or a string. |

**Source:**

The following inventory report indicates available quantities of product by product name. You need to verify that the product names are valid according to the following rules:

- A product name must begin with a three-digit numeric brand identifier, followed by a dash.
- A product name must end with a dash, followed by a six-digit numeric SKU.

Source data looks like the following, with the Validation column having no values in it.

| InvDate | ProductName | Qty | Validation |
|---------|-------------|-----|------------|
| 04/21/2017 | 412-Widgets-012345 | 23 | |
| 04/21/2017 | 04-Fidgets-120341 | 66 | |
| 04/21/2017 | 204-Midgets-4421 | 31 | |
| 04/21/2017 | 593-Gidgets-402012 | 24 | |

**Transformation:**

In this case, you must evaluate the `ProductName` column for two conditions. These conditional functions are the following:

```
IF(STARTSWITH(ProductName, `#{3}-`), 'Ok', 'Bad ProductName-Brand')
```

```
IF(ENDSWITH(ProductName, `-#{6}`), 'Ok', 'Bad ProductName-SKU')
```

One approach is to create two new test columns and then edit the column based on the evaluation of these two columns. However, using the following, you can compress the evaluation into a single step without creating the intermediate columns:

| Transformation Name | Edit column with formula |
|---------------------|--------------------------|
| Parameter: Columns | Status |
| Parameter: Formula | IF(STARTSWITH(ProductName, `#{3}-`), IF(ENDSWITH(ProductName, `-#{6}`), 'Ok', 'Bad ProductName-SKU'), 'Bad ProductName-Brand') |

**Results:**

| InvDate | ProductName | Qty | Validation |
|---------|-------------|-----|------------|

| | | | |
|---|---|---|---|
| 04/21/2017 | 412-Widgets-012345 | 23 | Ok |
| 04/21/2017 | 04-Fidgets-120341 | 66 | Bad ProductName-Brand |
| 04/21/2017 | 204-Midgets-4421 | 31 | Bad ProductName-SKU |
| 04/21/2017 | 593-Gidgets-402012 | 24 | Ok |

# REPEAT Function

**Contents:**

Repeats a string a specified number of times. The string can be specified as a String literal, a function returning a String, or a column reference.

- Since the `REPEAT` function matches based on fixed numeric values, changes to the length or structure of a data field can cause your recipe to fail to properly execute.
- The `REPEAT` function requires an integer value for the number of characters to match.

**Wrangle vs. SQL:** This function is part of Wrangle , a proprietary data transformation language. Wrangle is not SQL. For more information, see *Wrangle Language*.

## Basic Usage

**String literal example:**

```
repeat('ha',3)
```

**Output:** Returns the string: `hahaha`.

**Column reference example:**

```
repeat(MyString,4)
```

**Output:** Returns the values of the `MyString` column value written four times in a row.

## Syntax and Arguments

```
repeat(column_string,rpt_count)
```

| Argument | Required? | Data Type | Description |
|---|---|---|---|
| column_string | Y | string | Name of the column or string literal to be applied to the function |
| rpt_count | N | integer (positive) | Count of times to repeat the string |

For more information on syntax standards, see *Language Documentation Syntax Notes*.

**column_string**

Name of the column or String literal to be repeated.

- Missing string or column values generate missing string results.
- String constants must be quoted (`'Hello, World'`).
- Multiple columns and wildcards are not supported.

**Usage Notes:**

| Required? | Data Type | Example Value |
|-----------|-----------|---------------|
| Yes | String literal, function, or column reference | `myColumn` |

**rpt_count**

Count of times to repeat the string.

- If the value is not specified, the default is `1`.
- Value must a non-negative integer.
- References to columns of integer data type are not supported.

**Usage Notes:**

| Required? | Data Type | Example Value |
|-----------|-----------|---------------|
| No | Integer (non-negative) | 5 |

## Examples

> **Tip:** For additional examples, see *Common Tasks*.

**Example - REPEAT string function**

**Source:**

| myStr | repeat_count |
|-------|--------------|
| ha | 0 |
| ha | 1 |
| ha | 1.5 |
| ha | 2 |
| ha | -2 |

**Transformation:**

| Transformation Name | `New formula` |
|---------------------|---------------|
| **Parameter: Formula type** | `Single row formula` |
| **Parameter: Formula** | `repeat(myStr,repeat_count)` |
| **Parameter: New column name** | `'repeat_string'` |

**Results:**

| myStr | repeat_count | repeat_string |
|-------|--------------|---------------|
| ha    | 0            |               |
| ha    | 1            | ha            |
| ha    | 1.5          |               |
| ha    | 2            | haha          |
| ha    | -2           |               |

### Example - Padding values

In the following example, the imported `prodId` values are supposed to be eight characters in length. Somewhere during the process, however, leading `0` characters were truncated. The steps below allow you to re-insert the leading characters.

**Source:**

| prodName | prodId   |
|----------|----------|
| w01      | 1        |
| w02      | 10000001 |
| w03      | 345      |
| w04      | 10402    |

**Transformation:**

First, we need to identify how many zeroes need to be inserted for each prodId:

| Transformation Name | New formula |
|---------------------|-------------|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | 8 - len(prodId) |
| Parameter: New column name | 'len_prodId' |

Use the `REPEAT` function to generate a pad string based on the above values:

| Transformation Name | New formula |
|---------------------|-------------|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | repeat('0', len_prodId) |
| Parameter: New column name | 'padString' |

Merge the pad string and the original prodId column:

| Transformation Name | Merge columns |
|---------------------|---------------|
| Parameter: Columns | padString,prodId |

| Parameter: Separator | `' '` |
|---|---|
| Parameter: New column name | `'column2'` |

**Results:**

When you delete the intermediate columns and rename `column2` to `prodId`, you have the following table:

| prodName | prodId |
|---|---|
| w01 | 00000001 |
| w02 | 10000001 |
| w03 | 00000345 |
| w04 | 00010402 |

# EXACT Function

**Contents:**

- *Basic Usage*
- *Syntax and Arguments*
  - *string_ref1, string_ref2*
  - *ignore_case*
- *Examples*
  - *Example - Simple string comparisons*

---

Returns `true` if the second string evaluates to be an exact match of the first string. Source values can be string literals, column references, or expressions that evaluate to strings.

**Wrangle vs. SQL:** This function is part of Wrangle , a proprietary data transformation language. Wrangle is not SQL. For more information, see *Wrangle Language*.

## Basic Usage

**String literal reference example:**

```
exact('a','a')
```

**Output:** Returns `true`, since the values are identical.

**String literal reference example:**

```
exact('a','A')
```

**Output:** Returns `false`, since the capitalization is different between the two strings.

**Column reference example:**

```
exact(string1,string2)
```

**Output:** Returns the evaluation of `string1` column values being exact matches with the corresponding `string2` column values.

## Syntax and Arguments

```
exact(string_ref1,string_ref2 [,ignore_case])
```

| Argument | Required? | Data Type | Description |
|----------|-----------|-----------|-------------|
| string_ref1 | Y | string | Name of first column or first string literal to apply to the function |
| string_ref2 | Y | string | Name of second column or second string literal to apply to the function |
| ignore_case | N | string | When `true`, matching is case-insensitive. Default is `false`. |

For more information on syntax standards, see *Language Documentation Syntax Notes.*

### string_ref1, string_ref2

String literal, column reference, or expression whose elements you want to compare based on this function.

#### Usage Notes:

| Required? | Data Type | Example Value |
|---|---|---|
| Yes | String literal, column reference, or expression evaluating to a string | `myString1, myString2` |

### ignore_case

When `true`, matches are case-insensitive. Default is `false`.

> **NOTE:** This argument is not required. By default, matches are case-sensitive.

| Required? | Data Type | Example Value |
|---|---|---|
| No | String literal evaluating to a Boolean | `'true'` |

## Examples

> **Tip:** For additional examples, see *Common Tasks*.

### Example - Simple string comparisons

This example demonstrates functions that can be used to compare two sets of strings.

**Functions**:

| Item | Description |
|---|---|
| STRINGGREATERTHAN Function | Returns `true` if the first string evaluates to be greater than the second string, based on a set of common collation rules. |
| STRINGGREATERTHANEQUAL Function | Returns `true` if the first string evaluates to be greater than or equal to the second string, based on a set of common collation rules. |
| STRINGLESSTHAN Function | Returns `true` if the first string evaluates to be less than the second string, based on a set of common collation rules. |
| STRINGLESSTHANEQUAL Function | Returns `true` if the first string evaluates to be less than or equal to the second string, based on a set of common collation rules. |
| EXACT Function | Returns `true` if the second string evaluates to be an exact match of the first string. Source values can be string literals, column references, or expressions that evaluate to strings. |

#### Source:

The following table contains some example strings to be compared.

| rowId | stringA | stringB |
|---|---|---|

| | | |
|---|---|---|
| 1 | a | a |
| 2 | a | A |
| 3 | a | b |
| 4 | a | 1 |
| 5 | a | ; |
| 6 | ; | 1 |
| 7 | a | a |
| 8 | a | aa |
| 9 | abc | x |

Note that in row #6, `stringB` begins with a space character.

**Transformation:**

For each set of strings, the following functions are applied to generate a new column containing the results of the comparison.

| Transformation Name | New formula |
|---|---|
| **Parameter: Formula type** | Single row formula |
| **Parameter: Formula** | STRINGGREATERTHAN(stringA,stringB) |
| **Parameter: New column name** | 'greaterThan' |

| Transformation Name | New formula |
|---|---|
| **Parameter: Formula type** | Single row formula |
| **Parameter: Formula** | STRINGGREATERTHANEQUAL(stringA,stringB) |
| **Parameter: New column name** | 'greaterThanEqual' |

| Transformation Name | New formula |
|---|---|
| **Parameter: Formula type** | Single row formula |
| **Parameter: Formula** | STRINGLESSTHAN(stringA,stringB) |
| **Parameter: New column name** | 'lessThan' |

| Transformation Name | New formula |
|---|---|
| **Parameter: Formula type** | Single row formula |
| **Parameter: Formula** | STRINGLESSTHANEQUAL(stringA,stringB) |
| **Parameter: New column name** | 'lessThanEqual' |

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | `Single row formula` |
| Parameter: Formula | `EXACT(stringA,stringB)` |
| Parameter: New column name | `'exactEqual'` |

**Results:**

In the following table, the `Notes` column has been added manually.

| rowId | stringA | stringB | lessThanEqual | lessThan | greaterThanEqual | greaterThan | exactEqual | Notes |
|---|---|---|---|---|---|---|---|---|
| 1 | a | a | `true` | `false` | `true` | `false` | `true` | Evaluation of differences between `STRINGLESSTHAN` and `STRINGGREATERTHAN` and greater than versions. |
| 2 | a | A | `true` | `true` | `false` | `false` | `false` | Comparisons are case-sensitive. Uppercase letters are greater than lowercase letters. |
| 3 | a | b | `true` | `true` | `false` | `false` | `false` | Letters later in the alphabet (b) are greater than earlier letters (a). |
| 4 | a | 1 | `false` | `false` | `true` | `true` | `false` | Letters (a) are greater than digits (1). |
| 5 | a | ; | `false` | `false` | `true` | `true` | `false` | Letters (a) are greater than non-alphanumerics (;). |

| 6 | ; | 1 | true | true | false | false | false | Digits (1) are greater than non-alphanumerics (;). Therefore, the following characters are listed in order of evaluation:<br><br>`Aa1;` |
|---|---|---|------|------|-------|-------|-------|---|
| 7 | a | a | false | false | true | true | false | Letters (and any non-breaking character) are greater than space values. |
| 8 | a | aa | true | true | false | false | false | The second string is greater, since it contains one additional string at the end. |
| 9 | abc | x | true | true | false | false | false | The second string is greater, since its first letter is greater than the first letter of the first string. |

# STRINGGREATERTHAN Function

**Contents:**

Returns `true` if the first string evaluates to be greater than the second string, based on a set of common collation rules.

Source values can be string literals, column references, or expressions that evaluate to strings.

**Wrangle vs. SQL:** This function is part of Wrangle , a proprietary data transformation language. Wrangle is not SQL. For more information, see *Wrangle Language*.

## Basic Usage

**String literal reference example:**

```
stringgreaterthan('a','b')
```

**Output:** Returns `false` since `a` evaluates to be less than `b`.

**String literal reference example:**

```
stringgreaterthan('a','ab')
```

**Output:** Returns `false` since the second string contains an additional letter.

**String literal reference example:**

```
stringgreaterthan('abc','x')
```

**Output:** Returns `false` since the first letter of the first string is less than the first letter of the second string.

**Column reference example:**

```
stringgreaterthan(string1,string2)
```

**Output:** Returns the evaluation of `string1` column values being greater than `string2` column values.

**Collation** refers to the organizing of written content into a standardized order. String comparison functions utilize collation rules for Latin. A summary of the rules:

- Comparisons are case-sensitive.
    - Uppercase letters are greater than lowercase versions of the same letter.
    - However, lowercase letters that are later in the alphabet are greater than the uppercase version of the previous letter.
- Two strings are equal if they match identically.

- If two strings are identical except that the second string contains one additional character at the end, the second string is greater.
- A **normalized version** of a letter is the unaccented, lowercase version of the letter. In string comparison, it is the lowest value of all of its variants.
  - a is less than .
  - However, when compared to b, a = .
  - The set of Latin normalized characters contains more than 26 characters.

This table illustrates some generalized rules of Latin collation.

| Order | Description | Lesser Example | Greater Example |
|-------|-------------|----------------|-----------------|
| 1 | whitespace | (space) | (return) |
| 2 | Punctuation | ' | @ |
| 3 | Digits | 1 | 2 |
| 4 | Letters | a | A |
| 5 | | A | b |

**Resources:**

> **NOTE:** In the following set of charts (linked below), the values at the top of the page are lower than the values listed lower on the page. Similarly, the charts listed in the left nav bar are listed in ascending order.

For more information on the applicable collation rules, see *http://www.unicode.org/charts/collation/*.

## Syntax and Arguments

```
stringgreaterthan(string_ref1,string_ref2 <span>[,ignore_case]</span>)
```

| Argument | Required? | Data Type | Description |
|----------|-----------|-----------|-------------|
| string_ref1 | Y | string | Name of first column or first string literal to apply to the function |
| string_ref2 | Y | string | Name of second column or second string literal to apply to the function |
| ignore_case | N | string | When `true`, matching is case-insensitive. Default is `false`. |

For more information on syntax standards, see *Language Documentation Syntax Notes*.

**string_ref1, string_ref2**

String literal, column reference, or expression whose elements you want to compare based on this function.

**Usage Notes:**

| Required? | Data Type | Example Value |
|-----------|-----------|---------------|
| Yes | String literal, column reference, or expression evaluating to a string | `myString1, myString2` |

**ignore_case**

When `true`, matches are case-insensitive. Default is `false`.

> **NOTE:** This argument is not required. By default, matches are case-sensitive.

| Required? | Data Type | Example Value |
|-----------|-----------|---------------|
| No | String literal evaluating to a Boolean | `'true'` |

## Examples

> **Tip:** For additional examples, see *Common Tasks*.

### Example - Simple string comparisons

This example demonstrates functions that can be used to compare two sets of strings.

**Functions**:

| Item | Description |
|------|-------------|
| STRINGGREATERTHAN Function | Returns `true` if the first string evaluates to be greater than the second string, based on a set of common collation rules. |
| STRINGGREATERTHAN EQUAL Function | Returns `true` if the first string evaluates to be greater than or equal to the second string, based on a set of common collation rules. |
| STRINGLESSTHAN Function | Returns `true` if the first string evaluates to be less than the second string, based on a set of common collation rules. |
| STRINGLESSTHANEQU AL Function | Returns `true` if the first string evaluates to be less than or equal to the second string, based on a set of common collation rules. |
| EXACT Function | Returns `true` if the second string evaluates to be an exact match of the first string. Source values can be string literals, column references, or expressions that evaluate to strings. |

**Source:**

The following table contains some example strings to be compared.

| rowId | stringA | stringB |
|-------|---------|---------|
| 1 | a | a |
| 2 | a | A |
| 3 | a | b |
| 4 | a | 1 |
| 5 | a | ; |
| 6 | ; | 1 |

| 7 | a | a |
| 8 | a | aa |
| 9 | abc | x |

Note that in row #6, stringB begins with a space character.

**Transformation:**

For each set of strings, the following functions are applied to generate a new column containing the results of the comparison.

| Transformation Name | New formula |
|---|---|
| **Parameter: Formula type** | Single row formula |
| **Parameter: Formula** | STRINGGREATERTHAN(stringA,stringB) |
| **Parameter: New column name** | 'greaterThan' |

| Transformation Name | New formula |
|---|---|
| **Parameter: Formula type** | Single row formula |
| **Parameter: Formula** | STRINGGREATERTHANEQUAL(stringA,stringB) |
| **Parameter: New column name** | 'greaterThanEqual' |

| Transformation Name | New formula |
|---|---|
| **Parameter: Formula type** | Single row formula |
| **Parameter: Formula** | STRINGLESSTHAN(stringA,stringB) |
| **Parameter: New column name** | 'lessThan' |

| Transformation Name | New formula |
|---|---|
| **Parameter: Formula type** | Single row formula |
| **Parameter: Formula** | STRINGLESSTHANEQUAL(stringA,stringB) |
| **Parameter: New column name** | 'lessThanEqual' |

| Transformation Name | New formula |
|---|---|
| **Parameter: Formula type** | Single row formula |
| **Parameter: Formula** | EXACT(stringA,stringB) |
| **Parameter: New column name** | 'exactEqual' |

**Results:**

In the following table, the `Notes` column has been added manually.

| rowId | stringA | stringB | lessThanEqual | lessThan | greaterThanEqual | greaterThan | exactEqual | Notes |
|---|---|---|---|---|---|---|---|---|
| 1 | a | a | true | false | true | false | true | Evaluation of differences between `STRINGLESSTHAN` and `STRINGGREATERTHAN` and greater than versions. |
| 2 | a | A | true | true | false | false | false | Comparisons are case-sensitive. Uppercase letters are greater than lowercase letters. |
| 3 | a | b | true | true | false | false | false | Letters later in the alphabet (b) are greater than earlier letters (a). |
| 4 | a | 1 | false | false | true | true | false | Letters (a) are greater than digits (1). |
| 5 | a | ; | false | false | true | true | false | Letters (a) are greater than non-alphanumerics (;). |
| 6 | ; | 1 | true | true | false | false | false | Digits (1) are greater than non-alphanumerics (;). Therefore, the following characters are listed in order of evaluation:<br><br>`Aa1;` |
| 7 | a | a | false | false | true | true | false | Letters (and any non-breaking character) are greater than space values. |
| 8 | a | aa | true | true | false | false | false | The second string is greater, since it |

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | | | | | | | | contains one additional string at the end. |
| 9 | abc | x | true | true | false | false | false | The second string is greater, since its first letter is greater than the first letter of the first string. |

# STRINGGREATERTHANEQUAL Function

**Contents:**

---

Returns `true` if the first string evaluates to be greater than or equal to the second string, based on a set of common collation rules.

Source values can be string literals, column references, or expressions that evaluate to strings.

**Wrangle vs. SQL:** This function is part of Wrangle , a proprietary data transformation language. Wrangle is not SQL. For more information, see *Wrangle Language*.

## Basic Usage

**String literal reference example:**

```
stringgreaterthanequal(&apos;a&apos;,&apos;a&apos;)
```

**Output:** Returns `true` since both values are the same.

**String literal reference example:**

```
stringgreaterthanequal(&apos;a&apos;,&apos;b&apos;)
```

**Output:** Returns `false` since `a` evaluates to be less than `b`.

**String literal reference example:**

```
stringgreaterthanequal(&apos;abc&apos;,&apos;x&apos;)
```

**Output:** Returns `false` since the first letter of the first string is less than the first letter of the second string.

**Column reference example:**

```
stringgreaterthanequal(string1,string2)
```

**Output:** Returns the evaluation of `string1` column values being greater than `string2` column values.

**Collation** refers to the organizing of written content into a standardized order. String comparison functions utilize collation rules for Latin. A summary of the rules:

- Comparisons are case-sensitive.
    - Uppercase letters are greater than lowercase versions of the same letter.
    - However, lowercase letters that are later in the alphabet are greater than the uppercase version of the previous letter.

- Two strings are equal if they match identically.
    - If two strings are identical except that the second string contains one additional character at the end, the second string is greater.
- A **normalized version** of a letter is the unaccented, lowercase version of the letter. In string comparison, it is the lowest value of all of its variants.
    - a is less than .
    - However, when compared to b, a = .
    - The set of Latin normalized characters contains more than 26 characters.

This table illustrates some generalized rules of Latin collation.

| Order | Description | Lesser Example | Greater Example |
|-------|-------------|----------------|-----------------|
| 1 | whitespace | (space) | (return) |
| 2 | Punctuation | ' | @ |
| 3 | Digits | 1 | 2 |
| 4 | Letters | a | A |
| 5 | | A | b |

**Resources:**

> **NOTE:** In the following set of charts (linked below), the values at the top of the page are lower than the values listed lower on the page. Similarly, the charts listed in the left nav bar are listed in ascending order.

For more information on the applicable collation rules, see *http://www.unicode.org/charts/collation/*.

## Syntax and Arguments

```
stringgreaterthanequal(string_ref1,string_ref2 <span>[,ignore_case]</span>)
```

| Argument | Required? | Data Type | Description |
|----------|-----------|-----------|-------------|
| string_ref1 | Y | string | Name of first column or first string literal to apply to the function |
| string_ref2 | Y | string | Name of second column or second string literal to apply to the function |
| ignore_case | N | string | When `true`, matching is case-insensitive. Default is `false`. |

For more information on syntax standards, see *Language Documentation Syntax Notes*.

**string_ref1, string_ref2**

String literal, column reference, or expression whose elements you want to compare based on this function.

**Usage Notes:**

| Required? | Data Type | Example Value |
|-----------|-----------|---------------|
| Yes | String literal, column reference, or expression evaluating to a string | `myString1, myString2` |

**ignore_case**

When `true`, matches are case-insensitive. Default is `false`.

> **NOTE:** This argument is not required. By default, matches are case-sensitive.

| Required? | Data Type | Example Value |
|---|---|---|
| No | String literal evaluating to a Boolean | `'true'` |

## Examples

> **Tip:** For additional examples, see *Common Tasks*.

### Example - Simple string comparisons

This example demonstrates functions that can be used to compare two sets of strings.

**Functions**:

| Item | Description |
|---|---|
| STRINGGREATERTHAN Function | Returns `true` if the first string evaluates to be greater than the second string, based on a set of common collation rules. |
| STRINGGREATERTHAN EQUAL Function | Returns `true` if the first string evaluates to be greater than or equal to the second string, based on a set of common collation rules. |
| STRINGLESSTHAN Function | Returns `true` if the first string evaluates to be less than the second string, based on a set of common collation rules. |
| STRINGLESSTHANEQU AL Function | Returns `true` if the first string evaluates to be less than or equal to the second string, based on a set of common collation rules. |
| EXACT Function | Returns `true` if the second string evaluates to be an exact match of the first string. Source values can be string literals, column references, or expressions that evaluate to strings. |

**Source:**

The following table contains some example strings to be compared.

| rowId | stringA | stringB |
|---|---|---|
| 1 | a | a |
| 2 | a | A |
| 3 | a | b |
| 4 | a | 1 |
| 5 | a | ; |
| 6 | ; | 1 |

| 7 | a | a |
|---|---|---|
| 8 | a | aa |
| 9 | abc | x |

Note that in row #6, `stringB` begins with a space character.

**Transformation:**

For each set of strings, the following functions are applied to generate a new column containing the results of the comparison.

| **Transformation Name** | New formula |
|---|---|
| **Parameter: Formula type** | Single row formula |
| **Parameter: Formula** | STRINGGREATERTHAN(stringA,stringB) |
| **Parameter: New column name** | 'greaterThan' |

| **Transformation Name** | New formula |
|---|---|
| **Parameter: Formula type** | Single row formula |
| **Parameter: Formula** | STRINGGREATERTHANEQUAL(stringA,stringB) |
| **Parameter: New column name** | 'greaterThanEqual' |

| **Transformation Name** | New formula |
|---|---|
| **Parameter: Formula type** | Single row formula |
| **Parameter: Formula** | STRINGLESSTHAN(stringA,stringB) |
| **Parameter: New column name** | 'lessThan' |

| **Transformation Name** | New formula |
|---|---|
| **Parameter: Formula type** | Single row formula |
| **Parameter: Formula** | STRINGLESSTHANEQUAL(stringA,stringB) |
| **Parameter: New column name** | 'lessThanEqual' |

| **Transformation Name** | New formula |
|---|---|
| **Parameter: Formula type** | Single row formula |
| **Parameter: Formula** | EXACT(stringA,stringB) |
| **Parameter: New column name** | 'exactEqual' |

## Results:

In the following table, the `Notes` column has been added manually.

| rowId | stringA | stringB | lessThanEqual | lessThan | greaterThanEqual | greaterThan | exactEqual | Notes |
|---|---|---|---|---|---|---|---|---|
| 1 | a | a | true | false | true | false | true | Evaluation of differences between `STRINGLESSTHAN` and `STRINGGREATERTHAN` and greater than versions. |
| 2 | a | A | true | true | false | false | false | Comparisons are case-sensitive. Uppercase letters are greater than lowercase letters. |
| 3 | a | b | true | true | false | false | false | Letters later in the alphabet (b) are greater than earlier letters (a). |
| 4 | a | 1 | false | false | true | true | false | Letters (a) are greater than digits (1). |
| 5 | a | ; | false | false | true | true | false | Letters (a) are greater than non-alphanumerics (;). |
| 6 | ; | 1 | true | true | false | false | false | Digits (1) are greater than non-alphanumerics (;). Therefore, the following characters are listed in order of evaluation: `Aa1;` |
| 7 | a |   a | false | false | true | true | false | Letters (and any non-breaking character) are greater than space values. |
| 8 | a | aa | true | true | false | false | false | The second string is greater, since it |

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| | | | | | | | contains one additional string at the end. |
| 9 | abc | x | true | true | false | false | false | The second string is greater, since its first letter is greater than the first letter of the first string. |

# STRINGLESSTHAN Function

**Contents:**

- *Basic Usage*
- *Syntax and Arguments*
  - *string_ref1, string_ref2*
  - *ignore_case*
- *Examples*
  - *Example - Simple string comparisons*

---

Returns `true` if the first string evaluates to be less than the second string, based on a set of common collation rules.

Source values can be string literals, column references, or expressions that evaluate to strings.

**Wrangle vs. SQL:** This function is part of Wrangle , a proprietary data transformation language. Wrangle is not SQL. For more information, see *Wrangle Language*.

## Basic Usage

**String literal reference example:**

```
stringlessthan('a','b')
```

**Output:** Returns `true` since `a` evaluates to be less than `b`.

**String literal reference example:**

```
stringlessthan('a','ab')
```

**Output:** Returns `true` since the second string contains an additional letter.

**String literal reference example:**

```
stringlessthan('abc','x')
```

**Output:** Returns `true` since the first letter of the first string is less than the first letter of the second string.

**Column reference example:**

```
stringlessthan(string1,string2)
```

**Output:** Returns the evaluation of `string1` column values being greater than `string2` column values.

**Collation** refers to the organizing of written content into a standardized order. String comparison functions utilize collation rules for Latin. A summary of the rules:

- Comparisons are case-sensitive.
  - Uppercase letters are greater than lowercase versions of the same letter.
  - However, lowercase letters that are later in the alphabet are greater than the uppercase version of the previous letter.
- Two strings are equal if they match identically.

- If two strings are identical except that the second string contains one additional character at the end, the second string is greater.
- A **normalized version** of a letter is the unaccented, lowercase version of the letter. In string comparison, it is the lowest value of all of its variants.
  - a is less than .
  - However, when compared to b, a = .
  - The set of Latin normalized characters contains more than 26 characters.

This table illustrates some generalized rules of Latin collation.

| Order | Description | Lesser Example | Greater Example |
|-------|-------------|----------------|-----------------|
| 1 | whitespace | (space) | (return) |
| 2 | Punctuation | ' | @ |
| 3 | Digits | 1 | 2 |
| 4 | Letters | a | A |
| 5 | | A | b |

**Resources:**

> **NOTE:** In the following set of charts (linked below), the values at the top of the page are lower than the values listed lower on the page. Similarly, the charts listed in the left nav bar are listed in ascending order.

For more information on the applicable collation rules, see *http://www.unicode.org/charts/collation/*.

## Syntax and Arguments

```
stringlessthan(string_ref1,string_ref2 <span>[,ignore_case]</span>)
```

| Argument | Required? | Data Type | Description |
|----------|-----------|-----------|-------------|
| string_ref1 | Y | string | Name of first column or first string literal to apply to the function |
| string_ref2 | Y | string | Name of second column or second string literal to apply to the function |
| ignore_case | N | string | When `true`, matching is case-insensitive. Default is `false`. |

For more information on syntax standards, see *Language Documentation Syntax Notes*.

**string_ref1, string_ref2**

String literal, column reference, or expression whose elements you want to compare based on this function.

**Usage Notes:**

| Required? | Data Type | Example Value |
|-----------|-----------|---------------|
| Yes | String literal, column reference, or expression evaluating to a string | `myString1, myString2` |

**ignore_case**

When `true`, matches are case-insensitive. Default is `false`.

> **NOTE:** This argument is not required. By default, matches are case-sensitive.

| Required? | Data Type | Example Value |
|-----------|-----------|---------------|
| No | String literal evaluating to a Boolean | `'true'` |

## Examples

> **Tip:** For additional examples, see *Common Tasks*.

### Example - Simple string comparisons

This example demonstrates functions that can be used to compare two sets of strings.

**Functions**:

| Item | Description |
|------|-------------|
| STRINGGREATERTHAN Function | Returns `true` if the first string evaluates to be greater than the second string, based on a set of common collation rules. |
| STRINGGREATERTHAN EQUAL Function | Returns `true` if the first string evaluates to be greater than or equal to the second string, based on a set of common collation rules. |
| STRINGLESSTHAN Function | Returns `true` if the first string evaluates to be less than the second string, based on a set of common collation rules. |
| STRINGLESSTHANEQU AL Function | Returns `true` if the first string evaluates to be less than or equal to the second string, based on a set of common collation rules. |
| EXACT Function | Returns `true` if the second string evaluates to be an exact match of the first string. Source values can be string literals, column references, or expressions that evaluate to strings. |

**Source:**

The following table contains some example strings to be compared.

| rowId | stringA | stringB |
|-------|---------|---------|
| 1 | a | a |
| 2 | a | A |
| 3 | a | b |
| 4 | a | 1 |
| 5 | a | ; |
| 6 | ; | 1 |

| 7 | a | a |
|---|---|---|
| 8 | a | aa |
| 9 | abc | x |

Note that in row #6, `stringB` begins with a space character.

**Transformation:**

For each set of strings, the following functions are applied to generate a new column containing the results of the comparison.

| Transformation Name | New formula |
|---|---|
| **Parameter: Formula type** | Single row formula |
| **Parameter: Formula** | STRINGGREATERTHAN(stringA,stringB) |
| **Parameter: New column name** | 'greaterThan' |

| Transformation Name | New formula |
|---|---|
| **Parameter: Formula type** | Single row formula |
| **Parameter: Formula** | STRINGGREATERTHANEQUAL(stringA,stringB) |
| **Parameter: New column name** | 'greaterThanEqual' |

| Transformation Name | New formula |
|---|---|
| **Parameter: Formula type** | Single row formula |
| **Parameter: Formula** | STRINGLESSTHAN(stringA,stringB) |
| **Parameter: New column name** | 'lessThan' |

| Transformation Name | New formula |
|---|---|
| **Parameter: Formula type** | Single row formula |
| **Parameter: Formula** | STRINGLESSTHANEQUAL(stringA,stringB) |
| **Parameter: New column name** | 'lessThanEqual' |

| Transformation Name | New formula |
|---|---|
| **Parameter: Formula type** | Single row formula |
| **Parameter: Formula** | EXACT(stringA,stringB) |
| **Parameter: New column name** | 'exactEqual' |

**Results:**

In the following table, the `Notes` column has been added manually.

| rowId | stringA | stringB | lessThanEqual | lessThan | greaterThanEqual | greaterThan | exactEqual | Notes |
|---|---|---|---|---|---|---|---|---|
| 1 | a | a | true | false | true | false | true | Evaluation of differences between `STRINGLESSTHAN` and `STRINGGREATERTHAN` and greater than versions. |
| 2 | a | A | true | true | false | false | false | Comparisons are case-sensitive. Uppercase letters are greater than lowercase letters. |
| 3 | a | b | true | true | false | false | false | Letters later in the alphabet (b) are greater than earlier letters (a). |
| 4 | a | 1 | false | false | true | true | false | Letters (a) are greater than digits (1). |
| 5 | a | ; | false | false | true | true | false | Letters (a) are greater than non-alphanumerics (;). |
| 6 | ; | 1 | true | true | false | false | false | Digits (1) are greater than non-alphanumerics (;). Therefore, the following characters are listed in order of evaluation:<br><br>`Aa1;` |
| 7 | a |  a | false | false | true | true | false | Letters (and any non-breaking character) are greater than space values. |
| 8 | a | aa | true | true | false | false | false | The second string is greater, since it |

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | | | | | | | | contains one additional string at the end. |
| 9 | abc | x | true | true | false | false | false | The second string is greater, since its first letter is greater than the first letter of the first string. |

# STRINGLESSTHANEQUAL Function

**Contents:**

Returns `true` if the first string evaluates to be less than or equal to the second string, based on a set of common collation rules.

Source values can be string literals, column references, or expressions that evaluate to strings.

**Wrangle vs. SQL:** This function is part of Wrangle , a proprietary data transformation language. Wrangle is not SQL. For more information, see *Wrangle Language*.

## Basic Usage

**String literal reference example:**

```
stringlessthanequal('a','a')
```

**Output:** Returns `true` since both values are the same.

**String literal reference example:**

```
stringlessthanequal('a','b')
```

**Output:** Returns `true` since `a` evaluates to be less than `b`.

**String literal reference example:**

```
stringlessthanequal('abc','x')
```

**Output:** Returns `true` since the first letter of the first string is less than the first letter of the second string.

**Column reference example:**

```
stringlessthanequal(string1,string2)
```

**Output:** Returns the evaluation of `string1` column values being greater than `string2` column values.

**Collation** refers to the organizing of written content into a standardized order. String comparison functions utilize collation rules for Latin. A summary of the rules:

- Comparisons are case-sensitive.
    - Uppercase letters are greater than lowercase versions of the same letter.

- However, lowercase letters that are later in the alphabet are greater than the uppercase version of the previous letter.
- Two strings are equal if they match identically.
    - If two strings are identical except that the second string contains one additional character at the end, the second string is greater.
- A **normalized version** of a letter is the unaccented, lowercase version of the letter. In string comparison, it is the lowest value of all of its variants.
    - a is less than .
    - However, when compared to b, a = .
    - The set of Latin normalized characters contains more than 26 characters.

This table illustrates some generalized rules of Latin collation.

| Order | Description | Lesser Example | Greater Example |
|-------|-------------|----------------|-----------------|
| 1 | whitespace | (space) | (return) |
| 2 | Punctuation | ' | @ |
| 3 | Digits | 1 | 2 |
| 4 | Letters | a | A |
| 5 | | A | b |

**Resources:**

> **NOTE:** In the following set of charts (linked below), the values at the top of the page are lower than the values listed lower on the page. Similarly, the charts listed in the left nav bar are listed in ascending order.

For more information on the applicable collation rules, see *http://www.unicode.org/charts/collation/*.

## Syntax and Arguments

```
stringlessthanequal(string_ref1,string_ref2 <span>[,ignore_case]</span>)
```

| Argument | Required? | Data Type | Description |
|----------|-----------|-----------|-------------|
| string_ref1 | Y | string | Name of first column or first string literal to apply to the function |
| string_ref2 | Y | string | Name of second column or second string literal to apply to the function |
| ignore_case | N | string | When `true`, matching is case-insensitive. Default is `false`. |

For more information on syntax standards, see *Language Documentation Syntax Notes*.

**string_ref1, string_ref2**

String literal, column reference, or expression whose elements you want to compare based on this function.

**Usage Notes:**

| Required? | Data Type | Example Value |
|-----------|-----------|---------------|
| Yes | String literal, column reference, or expression evaluating to a string | `myString1, myString2` |

**ignore_case**

When `true`, matches are case-insensitive. Default is `false`.

> **NOTE:** This argument is not required. By default, matches are case-sensitive.

| Required? | Data Type | Example Value |
|---|---|---|
| No | String literal evaluating to a Boolean | `'true'` |

## Examples

> **Tip:** For additional examples, see *Common Tasks*.

### Example - Simple string comparisons

This example demonstrates functions that can be used to compare two sets of strings.

**Functions**:

| Item | Description |
|---|---|
| STRINGGREATERTHAN Function | Returns `true` if the first string evaluates to be greater than the second string, based on a set of common collation rules. |
| STRINGGREATERTHAN EQUAL Function | Returns `true` if the first string evaluates to be greater than or equal to the second string, based on a set of common collation rules. |
| STRINGLESSTHAN Function | Returns `true` if the first string evaluates to be less than the second string, based on a set of common collation rules. |
| STRINGLESSTHANEQU AL Function | Returns `true` if the first string evaluates to be less than or equal to the second string, based on a set of common collation rules. |
| EXACT Function | Returns `true` if the second string evaluates to be an exact match of the first string. Source values can be string literals, column references, or expressions that evaluate to strings. |

**Source:**

The following table contains some example strings to be compared.

| rowId | stringA | stringB |
|---|---|---|
| 1 | a | a |
| 2 | a | A |
| 3 | a | b |
| 4 | a | 1 |
| 5 | a | ; |
| 6 | ; | 1 |
| 7 | a | a |
| 8 | a | aa |

| 9 | abc | x |

Note that in row #6, `stringB` begins with a space character.

### Transformation:

For each set of strings, the following functions are applied to generate a new column containing the results of the comparison.

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | STRINGGREATERTHAN(stringA,stringB) |
| Parameter: New column name | 'greaterThan' |

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | STRINGGREATERTHANEQUAL(stringA,stringB) |
| Parameter: New column name | 'greaterThanEqual' |

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | STRINGLESSTHAN(stringA,stringB) |
| Parameter: New column name | 'lessThan' |

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | STRINGLESSTHANEQUAL(stringA,stringB) |
| Parameter: New column name | 'lessThanEqual' |

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | EXACT(stringA,stringB) |
| Parameter: New column name | 'exactEqual' |

### Results:

In the following table, the `Notes` column has been added manually.

| rowId | stringA | stringB | lessThanEqual | lessThan | greaterThanEqual | greaterThan | exactEqual | Notes |
|---|---|---|---|---|---|---|---|---|
| 1 | a | a | true | false | true | false | true | Evaluation of differences between `STRINGLESSTHAN` and `STRINGGREATERTHAN` and greater than versions. |
| 2 | a | A | true | true | false | false | false | Comparisons are case-sensitive. Uppercase letters are greater than lowercase letters. |
| 3 | a | b | true | true | false | false | false | Letters later in the alphabet (b) are greater than earlier letters (a). |
| 4 | a | 1 | false | false | true | true | false | Letters (a) are greater than digits (1). |
| 5 | a | ; | false | false | true | true | false | Letters (a) are greater than non-alphanumerics (;). |
| 6 | ; | 1 | true | true | false | false | false | Digits (1) are greater than non-alphanumerics (;). Therefore, the following characters are listed in order of evaluation: `Aa1;` |
| 7 | a |   a | false | false | true | true | false | Letters (and any non-breaking character) are greater than space values. |
| 8 | a | aa | true | true | false | false | false | The second string is greater, since it contains one additional string at the end. |

| 9 | abc | x | true | true | false | false | false | The second string is greater, since its first letter is greater than the first letter of the first string. |
|---|-----|---|------|------|-------|-------|-------|------|

# DOUBLEMETAPHONE Function

Returns a two-element array of primary and secondary phonetic encodings for an input string, based on the Double Metaphone algorithm.

The Double Metaphone algorithm processes an input string to render a primary and secondary spelling for it. For English language words, the algorithm removes silent letters, normalizes combinations of characters to a single definition, and removes vowels, except from the beginnings of words. In this manner, the algorithm can normalize inconsistencies between spellings for better matching. For more information, see *https://en.wikipedia.org/wiki/Metaphone*.

> **Tip:** This function is useful for performing fuzzy matching between string values, such as between potential join key values.

Source values can be string literals, column references, or expressions that evaluate to strings.

**Wrangle vs. SQL:** This function is part of Wrangle , a proprietary data transformation language. Wrangle is not SQL. For more information, see *Wrangle Language*.

## Basic Usage

**String literal reference example:**

```
doublemetaphone(&apos;My String&apos;)
```

**Output:** See below.

```
["MSTRNK","MSTRNK"]
```

### Column reference example:

```
doublemetaphone(string1)
```

**Output:** Generates a new `double_metaphone` column containing the evaluation of `string1` column values through the Double Metaphone algorithm.

## Syntax and Arguments

```
doublemetaphone(string_ref)
```

| Argument | Required? | Data Type | Description |
|----------|-----------|-----------|-------------|
| string_ref | Y | string | Name of column or string literal to apply to the function |

For more information on syntax standards, see *Language Documentation Syntax Notes*.

**string_ref1**

String literal, column reference, or expression whose elements you want to filter through the Double Metaphone algorithm.

**Usage Notes:**

| Required? | Data Type | Example Value |
|---|---|---|
| Yes | String literal, column reference, or expression evaluating to a string | `myString1` |

## Examples

> **Tip:** For additional examples, see *Common Tasks*.

### Example - Phonetic string comparisons

This example illustrates how to use double metaphone functions to generate phonetic spellings in Designer Cloud Powered by Trifacta® Enterprise Edition.

**Functions**:

| Item | Description |
|---|---|
| DOUBLEMETAPHONE Function | Returns a two-element array of primary and secondary phonetic encodings for an input string, based on the Double Metaphone algorithm. |
| DOUBLEMETAPHONEEQUALS Function | Compares two input strings using the Double Metaphone algorithm. An optional threshold parameter can be modified to adjust the tolerance for matching. |

**Source:**

The following table contains some example strings to be compared.

| string1 | string2 | notes |
|---|---|---|
| My String | my string | comparison is case-insensitive |
| judge | juge | typo |
| knock | nock | silent letters |
| white | wite | missing letters |
| record | record | two different words in English but match the same |
| pair | pear | these match but are different words. |
| bookkeeper | book keeper | spaces cause failures in comparison |
| test1 | test123 | digits are not compared |
| the end. | the end…. | punctuation differences do not matter. |
| a elephant | an elephant | a and an are treated differently. |

**Transformation:**

You can use the `DOUBLEMETAPHONE` function to generate phonetic spellings, as in the following:

| Transformation Name | New formula |
|---|---|
| **Parameter: Formula type** | Single row formula |
| | |

| Parameter: Formula | DOUBLEMETAPHONE(string1) |
|---|---|
| Parameter: New column name | 'dblmeta_s1' |

You can compare `string1` and `string2` using the `DOUBLEMETAPHONEEQUALS` function:

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | DOUBLEMETAPHONEEQUALS(string1, string2, 'normal') |
| Parameter: New column name | 'compare' |

### Results:

The following table contains some example strings to be compared.

| string1 | dblmeta_s1 | string2 | compare | Notes |
|---|---|---|---|---|
| My String | ["MSTRNK","MSTRNK"] | my string | TRUE | comparison is case-insensitive |
| judge | ["JJ","AJ"] | juge | TRUE | typo |
| knock | ["NK","NK"] | nock | TRUE | silent letters |
| white | ["AT","AT"] | wite | TRUE | missing letters |
| record | ["RKRT","RKRT"] | record | TRUE | two different words in English but match the same |
| pair | ["PR","PR"] | pear | TRUE | these match but are different words. |
| bookkeeper | ["PKPR","PKPR"] | book keeper | FALSE | spaces cause failures in comparison |
| test1 | ["TST","TST"] | test123 | TRUE | digits are not compared |
| the end. | ["0NT","TNT"] | the endâ€¦. | TRUE | punctuation differences do not matter. |
| a elephant | ["ALFNT","ALFNT"] | an elephant | FALSE | a and an are treated differently. |

# DOUBLEMETAPHONEEQUALS Function

**Contents:**

- *Basic Usage*
- *Syntax and Arguments*
    - *string_ref1, string_ref2*
    - *match_threshold*
- *Examples*
    - *Example - Phonetic string comparisons*

---

Compares two input strings using the Double Metaphone algorithm. An optional threshold parameter can be modified to adjust the tolerance for matching.

The Double Metaphone algorithm processes an input string render a primary and secondary spelling for it. For English language words, the algorithm removes silent letters, normalizes combinations of characters to a single definition, and removes vowels, except from the beginnings of words. In this manner, the algorithm can normalize inconsistencies between spellings for better matching. For more information, see *https://en.wikipedia.org/wiki/Metaphone*.

> **Tip:** This function is useful for performing fuzzy matching between string values, such as between potential join key values.

Source values can be string literals, column references, or expressions that evaluate to strings.

**Wrangle vs. SQL:** This function is part of Wrangle , a proprietary data transformation language. Wrangle is not SQL. For more information, see *Wrangle Language*.

## Basic Usage

**String literal reference example:**

```
doublemetaphoneequals('My String', 'my string')
```

**Output:** Returns the value `true`.

**Column reference example:**

```
doublemetaphoneequals(string1, string2, 'weak')
```

**Output:** Returns the comparison of `string1` and `string2` column values using the Double Metaphone algorithm. The `'weak'` parameter input means that only the secondary encodings for each input must match.

## Syntax and Arguments

```
doublemetaphoneequals(string_ref1, string_ref2, match_threshold)
```

| Argument | Required? | Data Type | Description |
|---|---|---|---|
| string_ref1 | Y | string | Name of first column or string literal to apply to the function |

| string_ref2 | Y | string | Name of second column or string literal to apply to the function |
|---|---|---|---|
| match_thresh old | N | string | Optional string value for the matching threshold to use in the comparison. Default value is `Normal`. |

For more information on syntax standards, see *Language Documentation Syntax Notes*.

### string_ref1, string_ref2

String literal, column reference, or expression whose elements you want to filter through the Double Metaphone algorithm.

### Usage Notes:

| Required? | Data Type | Example Value |
|---|---|---|
| Yes | String literal, column reference, or expression evaluating to a string | `myString1` |

### match_threshold

String literal identifying the threshold that determines a match according to the Double Metaphone encodings of the input strings. Accepted values:

| Threshold Value | Description |
|---|---|
| `'strong'` | The primary encodings of the two input strings must match. |
| `'normal'` | (Default) The primary encoding of one input string must match either of the encodings of the other input string. |
| `'weak'` | Either primary or secondary encoding of one input string must match either encoding of the other input string. |

### Usage Notes:

| Required? | Data Type | Example Value |
|---|---|---|
| Yes | String literal | `'strong'` |

## Examples

> **Tip:** For additional examples, see *Common Tasks*.

### Example - Phonetic string comparisons

This example illustrates how to use double metaphone functions to generate phonetic spellings in Designer Cloud Powered by Trifacta® Enterprise Edition.

**Functions**:

| Item | Description |
|---|---|
| DOUBLEMETAPHONE Function | Returns a two-element array of primary and secondary phonetic encodings for an input string, based on the Double Metaphone algorithm. |
| DOUBLEMETAPHONEEQU | Compares two input strings using the Double Metaphone algorithm. An optional threshold parameter can be |

| ALS Function | modified to adjust the tolerance for matching. |
|---|---|

**Source:**

The following table contains some example strings to be compared.

| string1 | string2 | notes |
|---|---|---|
| My String | my string | comparison is case-insensitive |
| judge | juge | typo |
| knock | nock | silent letters |
| white | wite | missing letters |
| record | record | two different words in English but match the same |
| pair | pear | these match but are different words. |
| bookkeeper | book keeper | spaces cause failures in comparison |
| test1 | test123 | digits are not compared |
| the end. | the end…. | punctuation differences do not matter. |
| a elephant | an elephant | a and an are treated differently. |

**Transformation:**

You can use the `DOUBLEMETAPHONE` function to generate phonetic spellings, as in the following:

| Transformation Name | `New formula` |
|---|---|
| **Parameter: Formula type** | `Single row formula` |
| **Parameter: Formula** | `DOUBLEMETAPHONE(string1)` |
| **Parameter: New column name** | `'dblmeta_s1'` |

You can compare `string1` and `string2` using the `DOUBLEMETAPHONEEQUALS` function:

| Transformation Name | `New formula` |
|---|---|
| **Parameter: Formula type** | `Single row formula` |
| **Parameter: Formula** | `DOUBLEMETAPHONEEQUALS(string1, string2, 'normal')` |
| **Parameter: New column name** | `'compare'` |

**Results:**

The following table contains some example strings to be compared.

| string1 | dblmeta_s1 | string2 | compare | Notes |
|---|---|---|---|---|
| My String | ["MSTRNK","MSTRNK"] | my string | TRUE | comparison is case-insensitive |
| judge | ["JJ","AJ"] | juge | TRUE | typo |
| knock | ["NK","NK"] | nock | TRUE | silent letters |
|  |  |  |  |  |

| white | ["AT","AT"] | wite | TRUE | missing letters |
|---|---|---|---|---|
| record | ["RKRT","RKRT"] | record | TRUE | two different words in English but match the same |
| pair | ["PR","PR"] | pear | TRUE | these match but are different words. |
| bookkeeper | ["PKPR","PKPR"] | book keeper | FALSE | spaces cause failures in comparison |
| test1 | ["TST","TST"] | test123 | TRUE | digits are not compared |
| the end. | ["0NT","TNT"] | the endâ€¦. | TRUE | punctuation differences do not matter. |
| a elephant | ["ALFNT","ALFNT"] | an elephant | FALSE | a and an are treated differently. |

# TRANSLITERATE Function

**Contents:**

Transliterates Asian script characters from one script form to another. The string can be specified as a column reference or a string literal.

**Wrangle vs. SQL:** This function is part of Wrangle , a proprietary data transformation language. Wrangle is not SQL. For more information, see *Wrangle Language*.

## Basic Usage

**Column reference example:**

```
transliterate(MyJapaneseSentences,HiraganaToKatakana)
```

**Output:** Returns the values in the `myJapaneseSentences` transliterated from Hiragana script form to Katakana script form.

## Syntax and Arguments

```
transliterate(column_string,form_enum)
```

| Argument | Required? | Data Type | Description |
|---|---|---|---|
| column_string | Y | string | Name of the column or string literal to be applied to the function |
| form_enum | Y | string (enumerated value) | The transliteration form as an enumerated value. Details below. |

For more information on syntax standards, see *Language Documentation Syntax Notes*.

**column_string**

Name of the column or string constant to be transliterated. String values must be in a supported Japanese script form. See below.

- Missing string or column values generate missing string results.
- String constants must be quoted (`'Hello, World'`).
- Multiple columns and wildcards are not supported.

**Usage Notes:**

| Required? | Data Type | Example Value |
|---|---|---|
|  |  |  |

| | | |
|---|---|---|
| Yes | String literal or column reference | `myColumn` |

### form_enum

Enumerated value to indicate the transliteration to apply to the referenced column:

> **NOTE:** Each width option can be paired with each form option. Four separate options are supported.

| Enum value | Description |
|---|---|
| `HiraganaToKatakana` | Transliterates Hiragana to Katagana |
| `KatakanaToHiragana` | Transliterates Katagana to Hiragana |
| `FullwidthToHalfwidth` | Transliterates full-width forms to half-width form |
| `HalfwidthToFullwidth` | Transliterates half-width forms to full-width form |

### Usage Notes:

| Required? | Data Type | Example Value |
|---|---|---|
| Yes | String (enumerated type) | `HiraganaToKatakana` |

## Examples

> **Tip:** For additional examples, see *Common Tasks*.

### Example - TRANSLITERATE Function

### Source:

| English | Japanese_Hiragana |
|---|---|
| a | |
| i | |
| u | |
| e | |
| o | |
| ka | |
| ki | |
| ku | |
| ke | |
| ko | |

### Transformation:

The following transliterates the above characters into Katakana form:

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | transliterate(Japanese_Hiragana, HiraganaToKatakana) |
| Parameter: New column name | 'Japanese_Katakana' |

The generated Katakana form is full-width. The following transliterates that column into half-width form:

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | transliterate(Japanese_Katakana, FullwidthToHalfwidth) |
| Parameter: New column name | 'Japanese_Katakana_halfwidth' |

**Results:**

| English | Japanese_Hiragana | Japanese_Katakana | Japanese_Katakana_halfwidth |
|---|---|---|---|
| a | | | |
| i | | | |
| u | | | |
| e | | | |
| o | | | |
| ka | | | |
| ki | | | |
| ku | | | |
| ke | | | |
| ko | | | |

# TRIMQUOTES Function

Removes leading and trailing quotes or double-quotes from a string. Quote marks in the middle of the string are not removed.

- This function applies to both single quotes (') and double quotes (").
  - This function is not limited to removing the outer set of quotes only. If there are multiple quotes at the beginning or the end of the string (""), all sets of quotes are removed.
- The `TRIMQUOTES` function does not remove whitespace at the beginning and end.

> **Tip:** You may need to nest this function and the `TRIM` function to clean up your strings. An example is provided below.

**Wrangle vs. SQL:** This function is part of Wrangle , a proprietary data transformation language. Wrangle is not SQL. For more information, see *Wrangle Language*.

## Basic Usage

**Column reference example:**

```
trimquotes(MyName)
```

**Output:** Returns the values of the `MyName` column value with any quotes removed from the beginning and the end.

**String literal examples:**

> **NOTE:** For string literal values that contain quotes to remove, you can bracket them in the quote mark of a different type. Some examples are below.

```
trimquotes(&apos;&quot;Hello, World&quot;&apos;)
```

**Output:** Input string is `"Hello, World"`. Output of the function is the string: `Hello, World`.

```
trimquotes(&apos;&quot;&quot;Hello,\&quot; World&quot;&quot;&apos;)
```

**Output:** Input string is `""Hello,\" World""`. Output of the function is the string: `Hello," World`.

Following input contains a single whitespace at the beginning and end of the string, which is the same as the previous string. To clean, you can first remove the whitespace with the `TRIM` function.

```
trimquotes(trim((&apos; &quot;&quot;Hello,\&quot; World&quot;&quot; &apos;)))
```

**Output:** Input string is `""Hello,\" World""` . Output of the function is the string: `Hello," World`. See *TRIM Function*.

## Syntax and Arguments

```
trimquotes(column_string)
```

| Argument | Required? | Data Type | Description |
|----------|-----------|-----------|-------------|
|          |           |           |             |

| | | | | |
|---|---|---|---|---|
| column_string | Y | | string | Name of the column or string literal to be applied to the function |

For more information on syntax standards, see *Language Documentation Syntax Notes*.

### column_string

Name of the column or string constant whose beginning and end quote marks are to be trimmed.

- Missing string or column values generate missing string results.
- String constants must be quoted (`'Hello, World'`).
- Multiple columns and wildcards are not supported.

**Usage Notes:**

| Required? | Data Type | Example Value |
|---|---|---|
| Yes | String literal or column reference | `myColumn` |

## Examples

> **Tip:** For additional examples, see *Common Tasks*.

### Example - String whitespace and quotes

The following example data contains a mixture of quotes and spaces strings. You can use the transformation listed below to clean up leading and trailing quotes and strings in a single transformation.

**Source:**

| String | Description |
|---|---|
| My String | "Base string: ""My String""" |
| My String extra | "Base string + "" extra""" |
| My String | A space in front of base string |
| My String | A space after base string |
| MyString | No space between the two words of base string |
| My  String | Two spaces between the two words of base string |
| "My String " | Base string + a tab character |
| "My String<br>" | Base string + a return character |
| "My String<br>" | Base string + a newline character |

**Transformation:**

You can use the following transformation which nests the `TRIM` and the `TRIMQUOTES` functions to clean up all of the columns in your dataset.

- To apply across all columns in the dataset:

- The wildcard (*) for columns indicates that this formula should be applied across all columns in the dataset.
  - You can also select `All` from the Columns drop-down in the Transform Builder.
  - Since all columns are String type, the results should be consistent.
- The `$col` reference can be used to refer to the current column that is being evaluated. For more information, see *Source Metadata References*.

| Transformation Name | `Edit column with formula` |
|---|---|
| **Parameter: Columns** | `All` |
| **Parameter: Formula** | `trimquotes(trim($col))` |

**Results:**

| String | Description |
|---|---|
| My String | Base string: ""My String |
| My String extra | Base string + "" extra |
| My String | A space in front of base string |
| My String | A space after base string |
| MyString | No space between the two words of base string |
| My  String | Two spaces between the two words of base string |
| My String | Base string + a tab character |
| My String | Base string + a return character |
| My String | Base string + a newline character |

# BASE64ENCODE Function

**Contents:**

Converts an input value to base64 encoding with optional padding with an equals sign (=). Input can be of any type. Output type is String.

- base64 is a method of representing data in a binary format over text protocols. During encoding, text values are converted to binary values 0-63. Each value is stored as an ASCII character based on a conversion chart.
    - Typically, base64 is used to transmit binary information, such as images, over transfer methods that use text, such as HTTP.

> **NOTE:** base64 is not an effective method of encryption.

- For more information on base64, see *https://en.wikipedia.org/wiki/Base64*.

**Wrangle vs. SQL:** This function is part of Wrangle , a proprietary data transformation language. Wrangle is not SQL. For more information, see *Wrangle Language*.

## Basic Usage

**Column reference example:**

```
base64encode(mySource)
```

**Output:** Returns the values from the `mySource` column written in base64 format.

**String literal example:**

```
base64encode('Hello, World. ', true)
```

**Output:** Returns the string: `GVsbG8sIFdvcmxkLiA=`. Note that the output string is padded with the equals sign at the end of the output value.

## Syntax and Arguments

```
base64encode(column_string,bool_padding)
```

| Argument | Required? | Data | Description |
|---|---|---|---|

| | | Type | |
|---|---|---|---|
| column_string | Y | string | Name of the column or string literal to be applied to the function |
| bool_padding | N | Boolean | When `true`, excess padding in the data stream is padded with an equals sign ( =) in the output. Default is `true`. |

For more information on syntax standards, see *Language Documentation Syntax Notes*.

### column_string

Name of the column or string constant to be converted.

- Missing string or column values generate missing string results.
- String constants must be quoted (`'Hello, World'`).
- Multiple columns and wildcards are not supported.

**Usage Notes:**

| Required? | Data Type | Example Value |
|---|---|---|
| Yes | String literal or column reference | `myColumn` |

### bool_padding

Boolean value that determines if spaces are padded with the equals sign.

Base64 represents six-bit values (0-63). These values are represented in encoded values as ASCII characters, which are 8-bit values (0-255).

For any arbitrary input, it is possible that the number of bits required to represent it as a base64 value (number of characters * 6) won't precisely match up ASCII representation. Four sextets of base64 encoding map to three octets of ASCII encoding. If the input string has been fully encoded, but there are extra ASCII octets so that the number of output octets is divisible by four.

When this parameter is set to `true`, the output value is padded with the equals sign (=) to represent output octets that are generated but do not contain any data encoded from the input. The default is `true`.

For more information on base64 padding, see *https://en.wikipedia.org/wiki/Base64*.

**Usage Notes:**

| Required? | Data Type | Example Value |
|---|---|---|
| No | Boolean | `false` |

## Examples

> **Tip:** For additional examples, see *Common Tasks*.

## Example - base64 encoding and decoding

This example demonstrates how to convert an input string to a base64-encoded value and back to ASCII text strings.

**Functions**:

| Item | Description |
|------|-------------|
| BASE64ENCODE Function | Converts an input value to base64 encoding with optional padding with an equals sign (=). Input can be of any type. Output type is String. |
| BASE64DECODE Function | Converts an input base64 value to text. Output type is String. |

### Source:

The following example contains three columns of different data types:

| IntegerField | StringField | ssn |
|--------------|-------------|-----|
| -2082863942 | This is a test string. | 987654321 |
| 2012994989 | "Hello, world." | 987654322 |
| -1637187918 | "Hello, world. Hello, world. Hello, world." | 987654323 |
| -1144194035 | fyi | 987654324 |
| -971872543 |  | 987654325 |
| 353977583 | This is a test string. | 987-65-4321 |
| -366583667 | "Hello, world." | 987-65-4322 |
| -573117553 | "Hello, world. Hello, world. Hello, world." | 987-65-4323 |
| 2051041970 | fyi | 987-65-4324 |
| 522691086 |  | 987-65-4325 |

### Transformation - encode:

You can use the following transformation to encode all of the columns in your dataset:

| | |
|---|---|
| **Transformation Name** | `Edit column with formula` |
| **Parameter: Columns** | `All` |
| **Parameter: Formula** | `base64encode($col, true)` |

### Results - encode:

The transformed dataset now looks like the following. Note the padding (equals signs) at the end of some of the values. Padding is added by default.

| IntegerField | StringField | ssn |
|--------------|-------------|-----|
| LTIwODI4NjM5NDI= | VGhpcyBpcyBhIHRlc3Qgc3RyaW5nLg== | OTg3NjU0MzIx |
| MjAxMjk5NDk4OQ== | IkhlbGxvLCB3b3JsZC4i | OTg3NjU0MzIy |
| LTE2MzcxODc5MTg= | IkhlbGxvLCB3b3JsZC4gSGVsbG8sIHdvcmxkLiBIZWxsbywgd29ybGQuIg== | OTg3NjU0MzIz |
|  |  |  |

| | | |
|---|---|---|
| LTExNDQxOTQwMzU= | Znlp | OTg3NjU0MzI0 |
| LTk3MTg3MjU0Mw== | | OTg3NjU0MzI1 |
| MzUzOTc3NTgz | VGhpcyBpcyBhIHRlc3Qgc3RyaW5nLg== | OTg3LTY1LTQzMjE= |
| LTM2NjU4MzY2Nw== | IkhlbGxvLCB3b3JsZC4i | OTg3LTY1LTQzMjI= |
| LTU3MzExNzU1Mw== | IkhlbGxvLCB3b3JsZC4gSGVsbG8sIHdvcmxkLiBIZWxsbywgd29ybGQuIg== | OTg3LTY1LTQzMjM= |
| MjA1MTA0MTk3MA== | Znlp | OTg3LTY1LTQzMjQ= |
| NTIyNjkxMDg2 | | OTg3LTY1LTQzMjU= |

### Transformation - decode:

The following transformation can be used to decode all of the columns:

| | |
|---|---|
| **Transformation Name** | `Edit column with formula` |
| **Parameter: Columns** | `All` |
| **Parameter: Formula** | `base64decode($col)` |

### Results - decode:

| IntegerField | StringField | ssn |
|---|---|---|
| -2082863942 | This is a test string. | 987654321 |
| 2012994989 | "Hello, world." | 987654322 |
| -1637187918 | "Hello, world. Hello, world. Hello, world." | 987654323 |
| -1144194035 | fyi | 987654324 |
| -971872543 | | 987654325 |
| 353977583 | This is a test string. | 987-65-4321 |
| -366583667 | "Hello, world." | 987-65-4322 |
| -573117553 | "Hello, world. Hello, world. Hello, world." | 987-65-4323 |
| 2051041970 | fyi | 987-65-4324 |
| 522691086 | | 987-65-4325 |

# BASE64DECODE Function

Converts an input base64 value to text. Output type is String.

- base64 is a method of representing data in a binary format over text protocols. During encoding, text values are converted to binary values 0-63. Each value is stored as an ASCII character based on a conversion chart.
  - Typically, base64 is used to transmit binary information, such as images, over transfer methods that use text, such as HTTP.

> **NOTE:** base64 is not an effective method of encryption.

- For more information on base64, see *https://en.wikipedia.org/wiki/Base64*.

**Wrangle vs. SQL:** This function is part of Wrangle , a proprietary data transformation language. Wrangle is not SQL. For more information, see *Wrangle Language*.

## Basic Usage

**Column reference example:**

```
base64decode(mySource)
```

**Output:** Decodes the base64 values from the `mySource` column into text.

**String literal example:**

```
base64decode(&apos;GVsbG8sIFdvcmxkLiA=&apos;)
```

**Output:** Decodes the input value to the following text: `Hello, World. .`

## Syntax and Arguments

```
base64decode(column_string)
```

| Argument | Required? | Data Type | Description |
|---|---|---|---|
| column_string | Y | string | Name of the column or string literal to be applied to the function |

For more information on syntax standards, see *Language Documentation Syntax Notes*.

**column_string**

Name of the column or string constant to be converted.

- Missing string or column values generate missing string results.
- String constants must be quoted (`'Hello, World'`).
- Multiple columns and wildcards are not supported.

**Usage Notes:**

| Required? | Data Type | Example Value |
|---|---|---|
| Yes | String literal or column reference | `myColumn` |

## Examples

> **Tip:** For additional examples, see *Common Tasks*.

### Example - base64 encoding and decoding

This example demonstrates how to convert an input string to a base64-encoded value and back to ASCII text strings.

**Functions**:

| Item | Description |
|---|---|
| BASE64ENCODE Function | Converts an input value to base64 encoding with optional padding with an equals sign (=). Input can be of any type. Output type is String. |
| BASE64DECODE Function | Converts an input base64 value to text. Output type is String. |

**Source:**

The following example contains three columns of different data types:

| IntegerField | StringField | ssn |
|---|---|---|
| -2082863942 | This is a test string. | 987654321 |
| 2012994989 | "Hello, world." | 987654322 |
| -1637187918 | "Hello, world. Hello, world. Hello, world." | 987654323 |
| -1144194035 | fyi | 987654324 |
| -971872543 | | 987654325 |
| 353977583 | This is a test string. | 987-65-4321 |
| -366583667 | "Hello, world." | 987-65-4322 |
| -573117553 | "Hello, world. Hello, world. Hello, world." | 987-65-4323 |
| 2051041970 | fyi | 987-65-4324 |
| 522691086 | | 987-65-4325 |

**Transformation - encode:**

You can use the following transformation to encode all of the columns in your dataset:

| | |
|---|---|
| **Transformation Name** | `Edit column with formula` |
| **Parameter: Columns** | `All` |
| | |

| Parameter: Formula | base64encode($col, true) |
|---|---|

### Results - encode:

The transformed dataset now looks like the following. Note the padding (equals signs) at the end of some of the values. Padding is added by default.

| IntegerField | StringField | ssn |
|---|---|---|
| LTIwODI4NjM5NDI= | VGhpcyBpcyBhIHRlc3Qgc3RyaW5nLg== | OTg3NjU0MzIx |
| MjAxMjk5NDk4OQ== | IkhlbGxvLCB3b3JsZC4i | OTg3NjU0MzIy |
| LTE2MzcxODc5MTg= | IkhlbGxvLCB3b3JsZC4gSGVsbG8sIHdvcmxkLiBIZWxsbywgd29ybGQuIg== | OTg3NjU0MzIz |
| LTExNDQxOTQwMzU= | Znlp | OTg3NjU0MzI0 |
| LTk3MTg3MjU0Mw== | | OTg3NjU0MzI1 |
| MzUzOTc3NTgz | VGhpcyBpcyBhIHRlc3Qgc3RyaW5nLg== | OTg3LTY1LTQzMjE= |
| LTM2NjU4MzY2Nw== | IkhlbGxvLCB3b3JsZC4i | OTg3LTY1LTQzMjI= |
| LTU3MzExNzU1Mw== | IkhlbGxvLCB3b3JsZC4gSGVsbG8sIHdvcmxkLiBIZWxsbywgd29ybGQuIg== | OTg3LTY1LTQzMjM= |
| MjA1MTA0MTk3MA== | Znlp | OTg3LTY1LTQzMjQ= |
| NTIyNjkxMDg2 | | OTg3LTY1LTQzMjU= |

### Transformation - decode:

The following transformation can be used to decode all of the columns:

| Transformation Name | Edit column with formula |
|---|---|
| Parameter: Columns | All |
| Parameter: Formula | base64decode($col) |

### Results - decode:

| IntegerField | StringField | ssn |
|---|---|---|
| -2082863942 | This is a test string. | 987654321 |
| 2012994989 | "Hello, world." | 987654322 |
| -1637187918 | "Hello, world. Hello, world. Hello, world." | 987654323 |
| -1144194035 | fyi | 987654324 |
| -971872543 | | 987654325 |
| 353977583 | This is a test string. | 987-65-4321 |
| -366583667 | "Hello, world." | 987-65-4322 |
| -573117553 | "Hello, world. Hello, world. Hello, world." | 987-65-4323 |
| 2051041970 | fyi | 987-65-4324 |
| 522691086 | | 987-65-4325 |

# Nested Functions

Nested functions can be used to modify or combine nested data types. For data of Object or Array type, these functions are valuable for transforming those columns with a minimum of steps.

> **NOTE:** Some functions apply to either Object type or Array type, not both.

# ARRAYCONCAT Function

Combines the elements of one array with another, listing all elements of the first array before listing all elements of the second array.

- Arrays are referenced by column name or as array literals.
- This function applies two or more columns of Array type only. To concatenate string values, see *Merge Transform*.
- Duplicate values are not removed from the generated array.

**Wrangle vs. SQL:** This function is part of Wrangle , a proprietary data transformation language. Wrangle is not SQL. For more information, see *Wrangle Language*.

## Basic Usage

**Array literal reference example:**

```
arrayconcat([["A","B","C"],["C","D",
"E"]])
```

**Output:** Generates the following array:

```
["A","B","C","C","D","E"]
```

**Column reference example:**

```
arrayconcat([array1,array2])
```

**Output:** Generates a new array containing a single array listing all of the elements in `array1` followed by all elements from `array2` in order.

## Syntax and Arguments

```
arrayconcat(array_ref1,array_ref2)
```

| Argument | Required? | Data Type | Description |
|---|---|---|---|
| array_ref1 | Y | string or array | Name of first column or first array literal to apply to the function |
| array_ref2 | Y | string or array | Name of second column or second array literal to apply to the function |

For more information on syntax standards, see *Language Documentation Syntax Notes*.

**array_ref1, array_ref2**

Array literal or name of the array column whose elements you want to concatenate together. You can conc atenat e together two or more arrays.

**Usage Notes:**

| Required? | Data Type | Example Value |
|---|---|---|

| Yes | Array literal or column reference | `myArray1,myArray2` |
|-----|-----------------------------------|---------------------|

## Examples

> **Tip:** For additional examples, see *Common Tasks*.

### Example - Simple concat example

This example illustrates how to use the nested functions.

**Functions**:

| Item | Description |
|------|-------------|
| ARRAYCONCAT Function | Combines the elements of one array with another, listing all elements of the first array before listing all elements of the second array. |
| ARRAYINTERSECT Function | Generates an array containing all elements that appear in multiple input arrays, referenced as column names or array literals. |
| ARRAYCROSS Function | Generates a nested array containing the cross-product of all elements in two or more arrays. |
| ARRAYUNIQUE Function | Generates an array of all unique elements among one or more arrays. |

**Source:**

Code formatting has been applied to improve legibility.

| Item | ArrayA | ArrayB |
|------|--------|--------|
| Item1 | `["A","B","C"]` | `["1","2","3"]` |
| Item2 | `["A","B"]` | `["A","B","C"]` |
| Item3 | `["D","E","F"]` | `["4","5","6"]` |

**Transformation:**

You can apply the following transforms in the following order. Note that the column names must be different from the transform name, which is a reserved word.

| Transformation Name | `New formula` |
|---------------------|---------------|
| Parameter: Formula type | `Single row formula` |
| Parameter: Formula | `ARRAYCONCAT([Letters,Numerals])` |
| Parameter: New column name | `'concat2'` |

| Transformation Name | `New formula` |
|---------------------|---------------|
| Parameter: Formula type | `Single row formula` |
| Parameter: Formula | `ARRAYINTERSECT([Letters,Numerals])` |
| Parameter: New column | `'intersection2'` |

| name | |
|------|---|

| Transformation Name | New formula |
|---------------------|-------------|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | ARRAYCROSS([Letters,Numerals]) |
| Parameter: New column name | 'cross2' |

| Transformation Name | New formula |
|---------------------|-------------|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | ARRAYUNIQUE([Letters,Numerals]) |
| Parameter: New column name | 'unique2' |

**Results:**

For display purposes, the results table has been broken down into three separate sets of columns.

Column set 1:

| Item | ArrayA | ArrayB | concat2 | intersection2 |
|------|--------|--------|---------|---------------|
| Item1 | ["A","B","C"] | ["1","2","3"] | ["A","B","C","1","2","3"] | [ ] |
| Item2 | ["A","B"] | ["A","B","C"] | ["A","B","A","B","C"] | ["A","B"] |
| Item3 | ["D","E","F"] | ["4","5","6"] | ["D","E","F","4","5","6"] | [ ] |

Column set 2:

| Item | cross2 |
|------|--------|
| Item1 | [["A","1"],["A","2"],["A","3"],["B","1"],["B","2"],["B","3"],["C","1"],["C","2"],["C","3"]] |
| Item2 | [["A","A"],["A","B"],["A","C"],["B","A"],["B","B"],["B","C"]] |
| Item3 | [["D","4"],["D","5"],["D","6"],["E","4"],["E","5"],["E","6"], ["F","4"],["F","5"],["F","6"]] |

Column set 3:

| Item | unique2 |
|------|---------|
| Item1 | ["A","B","C","1","2","3"] |
| Item2 | ["A","B","C"] |
| Item3 | ["D","E","F","4","5","6"] |

# ARRAYCROSS Function

Generates a nested array containing the cross-product of all elements in two or more arrays.

- Input arrays can be referenced as column names or array literals.
- If Array1 has M elements and Array2 has N elements, the generated array has M X N elements.

> **NOTE:** Be careful applying this function across columns of large arrays. A limit is automatically applied on large arrays to prevent overloading the browser. Avoid apply the `ARRAYCROSS` transform to very wide columns.

**Wrangle vs. SQL:** This function is part of Wrangle , a proprietary data transformation language. Wrangle is not SQL. For more information, see *Wrangle Language*.

## Basic Usage

**Array literal reference example:**

```
arraycross[["A","B"],["1","2","3"]])
```

**Output:** Returns a single array:

```
[["A","1"],["A","2"],["A","3"],["B","1"],["B","2"],["B","3"]]
```

**Column reference example:**

```
arraycross(array1,array2,array3)
```

**Output:** Returns an array containing a single array listing all combinations of elements between `array1`, `array2`, and `array3`.

## Syntax and Arguments

```
arraycross(array_ref1,array_ref2)
```

| Argument | Required? | Data Type | Description |
|---|---|---|---|
| array_ref1 | Y | string or array | Name of first column or first array literal to apply to the function |
| array_ref2 | Y | string or array | Name of second column or second array literal to apply to the function |

For more information on syntax standards, see *Language Documentation Syntax Notes*.

**array_ref1, array_ref2**

Array literal or name of the array column whose intersection you want to derive.

**Usage Notes:**

| Required? | Data Type | Example Value |
|---|---|---|
| Yes | Array literal or column reference | |

| | | myArray1,myArray2 |
|---|---|---|

## Examples

> **Tip:** For additional examples, see *Common Tasks*.

### Example - Simple cross example

This example illustrates how to use the nested functions.

**Functions**:

| Item | Description |
|---|---|
| ARRAYCONCAT Function | Combines the elements of one array with another, listing all elements of the first array before listing all elements of the second array. |
| ARRAYINTERSECT Function | Generates an array containing all elements that appear in multiple input arrays, referenced as column names or array literals. |
| ARRAYCROSS Function | Generates a nested array containing the cross-product of all elements in two or more arrays. |
| ARRAYUNIQUE Function | Generates an array of all unique elements among one or more arrays. |

**Source:**

Code formatting has been applied to improve legibility.

| Item | ArrayA | ArrayB |
|---|---|---|
| Item1 | ["A","B","C"] | ["1","2","3"] |
| Item2 | ["A","B"] | ["A","B","C"] |
| Item3 | ["D","E","F"] | ["4","5","6"] |

**Transformation:**

You can apply the following transforms in the following order. Note that the column names must be different from the transform name, which is a reserved word.

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | ARRAYCONCAT([Letters,Numerals]) |
| Parameter: New column name | 'concat2' |

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | ARRAYINTERSECT([Letters,Numerals]) |
| Parameter: New column | 'intersection2' |

| | name | |
|---|---|---|

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | ARRAYCROSS([Letters,Numerals]) |
| Parameter: New column name | 'cross2' |

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | ARRAYUNIQUE([Letters,Numerals]) |
| Parameter: New column name | 'unique2' |

**Results:**

For display purposes, the results table has been broken down into three separate sets of columns.

Column set 1:

| Item | ArrayA | ArrayB | concat2 | intersection2 |
|---|---|---|---|---|
| Item1 | ["A","B","C"] | ["1","2","3"] | ["A","B","C","1","2","3"] | [ ] |
| Item2 | ["A","B"] | ["A","B","C"] | ["A","B","A","B","C"] | ["A","B"] |
| Item3 | ["D","E","F"] | ["4","5","6"] | ["D","E","F","4","5","6"] | [ ] |

Column set 2:

| Item | cross2 |
|---|---|
| Item1 | [["A","1"],["A","2"],["A","3"],["B","1"],["B","2"],["B","3"],["C","1"],["C","2"],["C","3"]] |
| Item2 | [["A","A"],["A","B"],["A","C"],["B","A"],["B","B"],["B","C"]] |
| Item3 | [["D","4"],["D","5"],["D","6"],["E","4"],["E","5"],["E","6"], ["F","4"],["F","5"],["F","6"]] |

Column set 3:

| Item | unique2 |
|---|---|
| Item1 | ["A","B","C","1","2","3"] |
| Item2 | ["A","B","C"] |
| Item3 | ["D","E","F","4","5","6"] |

# ARRAYELEMENTAT Function

**Contents:**

Computes the 0-based index value for an array element in the specified column, array literal, or function that returns an array.

- This function calculates based on the outer layer of an array. If your array is nested, the count of inner elements is not factored.
- If a row contains a missing array, the returned value is `0`. If it contains a value that is not recognized as an array, the returned value is null.

**Wrangle vs. SQL:** This function is part of Wrangle , a proprietary data transformation language. Wrangle is not SQL. For more information, see *Wrangle Language*.

## Basic Usage

**Array literal reference example:**

```
arrayelementat([A,B,C,D],2)
```

**Output:** Returns the third value in the array, which is `C`.

**Column reference example:**

```
arrayelementat(myArrays,9)
```

**Output:** Returns the tenth element of the arrays listed in the `myArrays` column.

**Array function example:**

```
arrayelementat(concat([colA,colB]),3)
```

**Output:** Returns the fourth element of the concatenated array.

## Syntax and Arguments

```
arrayelementat(array_ref,int_index_ref)
```

| Argument | Required? | Data Type | Description |
|----------|-----------|-----------|-------------|
| array_ref | Y | string | Name of Array column, Array literal, or function returning an Array to apply to the function |

| | | | |
|---|---|---|---|
| int_index_ref | Y | integer (non-negative) | Index value for the array element to return. Value can be Integer literal, column containing Integer values, or function returning an Integer. |

For more information on syntax standards, see *Language Documentation Syntax Notes*.

### array_ref

Name of the array column, array literal, or function returning an array whose elements you want to return.

- Multiple columns and wildcards are not supported.

**Usage Notes:**

| Required? | Data Type | Example Value |
|---|---|---|
| Yes | String (column reference or function) or array literal | `myArray1` |

### int_index_ref

Non-negative integer value representing the index value of the array element to return. Value can be Integer literal, column containing Integer values, or function returning an Integer.

- Value must a non-negative integer. If the value is `0`, then the first element of the array is returned.
- If this value is greater than the length of the string, then a null value is returned.

**Usage Notes:**

| Required? | Data Type | Example Value |
|---|---|---|
| Yes | Integer (non-negative) | 5 |

## Examples

> **Tip:** For additional examples, see *Common Tasks*.

### Example - Student progress across tests

This example illustrates how to return n-based number of elements in an array.

**Functions**:

| Item | Description |
|---|---|
| ARRAYLEN Function | Computes the number of elements in the arrays in the specified column, array literal, or function that returns an array. |
| ARRAYELEMENTAT Function | Computes the 0-based index value for an array element in the specified column, array literal, or function that returns an array. |
| ARRAYSORT Function | Sorts array values in the specified column, array literal, or function that returns an array in ascending or descending order. |

**Source:**

Here are some student test scores. Individual scores are stored in the `Scores` column. You want to:

1. Flag the students who have not taken four tests.
2. Compute the range in scores for each student.

| LastName | FirstName | Scores |
|---|---|---|
| Allen | Amanda | [79, 83,87,81] |
| Bell | Bobby | [85, 92, 94, 98] |
| Charles | Cameron | [88,81,85] |
| Dudley | Danny | [82,88,81,77] |
| Ellis | Evan | [91,93,87,93] |

**Transformation:**

First, you want to flag the students who did not take all four tests:

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | IF(ARRAYLEN(Scores) < 4,"incomplete","") |
| Parameter: New column name | 'Error' |

This test flags Cameron Charles only.

The following transform sorts the array values in highest to lowest score:

| Transformation Name | Edit column with formula |
|---|---|
| Parameter: Columns | Scores |
| Parameter: Formula | ARRAYSORT(Scores, 'descending') |

The following transforms extracts the first (highest) and last (lowest) value in each student's test scores, provided that they took four tests:

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | ARRAYELEMENTAT(Scores,0) |
| Parameter: New column name | 'highestScore' |

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | ARRAYELEMENTAT(Scores,3) |

| Parameter: New column name | 'lowestScore' |
|---|---|

> **Tip:** You could also generate the `Error` column when the `Scores4` column contains a null value. If no value exists in the array for the `ARRAYELEMENTAT` function, a null value is returned, which would indicate in this case an insufficient number of elements (test scores).

You can now track change in test scores:

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | SUBTRACT(highestScore,lowestScore) |
| Parameter: New column name | 'Score_range' |

**Results:**

| LastName | FirstName | Scores | Error | lowestScore | highestScore | Score_range |
|---|---|---|---|---|---|---|
| Allen | Amanda | [87,83,81,79] | | 79 | 87 | 8 |
| Bell | Bobby | [98,94,92,85] | | 85 | 98 | 13 |
| Charles | Cameron | [88,85,81] | incomplete | | 88 | |
| Dudley | Danny | [88,82,81,77] | | 77 | 88 | 11 |
| Ellis | Evan | [93,93,91,87] | | 87 | 93 | 6 |

# ARRAYINDEXOF Function

**Contents:**

---

Computes the index at which a specified element is first found within an array. Indexing is left to right.

- Leftmost index value is `0`.
- If the element is not found, null is returned.
- For right-to-left searching, use ARRAYRIGHTINDEXOF.
    - If only one element exists in the array, both functions return the same value.
    - For more information, see *ARRAYRIGHTINDEXOF Function*.

**Wrangle vs. SQL:** This function is part of Wrangle , a proprietary data transformation language. Wrangle is not SQL. For more information, see *Wrangle Language*.

## Basic Usage

**Array literal reference example:**

```
arrayindexof(["A","B","C","D"],"C")
```

**Output:** Returns the index of the element "C" in the array, which is `2` in an 0-based index.

**Column reference example:**

```
arrayindexof([myValues],myElement)
```

**Output:** Returns the index in the `myValues` arrays for the elements listed in the `myElement` column.

## Syntax and Arguments

```
arrayindexof(array_ref,my_element)
```

| Argument | Required? | Data Type | Description |
|---|---|---|---|
| array_ref | Y | array or string | Name of Array column, Array literal, or function returning an Array to apply to the function |
| my_element | Y | any | The element to locate in the array |

For more information on syntax standards, see *Language Documentation Syntax Notes*.

**array_ref**

Name of the array column, array literal, or function returning an array whose element you want to locate.

- Multiple columns and wildcards are not supported.

**Usage Notes:**

| Required? | Data Type | Example Value |
|-----------|-----------|---------------|
| Yes | String (column reference or function) or array literal | `myArray1` |

**my_element**

Element literal that you wish to locate in the array. It can be a value of any data type.

**Usage Notes:**

| Required? | Data Type | Example Value |
|-----------|-----------|---------------|
| Yes | Any | `"1st"` |

## Examples

> **Tip:** For additional examples, see *Common Tasks*.

**Example - Computing points based on position of finish**

This example illustrates how to convert the index value of an array for a specified value searching from left to right and right to left by using ARRAYINDEXOF and ARRAYRIGTHINDEXOF functions.

**Functions**:

| Item | Description |
|------|-------------|
| ARRAYINDEXOF Function | Computes the index at which a specified element is first found within an array. Indexing is left to right. |
| ARRAYRIGHTINDEXOF Function | Computes the index at which a specified element is first found within an array, when searching right to left. Returned value is based on left-to-right indexing. |

**Source:**

The following set of arrays contain results, in order, of a series of races. From this list, the goal is to generate the score for each racer according to the following scoring matrix.

| Place | Points |
|-------|--------|
| 1st | 30 |
| 2nd | 20 |
| 3rd | 10 |

| Last | -10 |
| --- | --- |
| Did Not Finish (DNF) | -20 |

Results:

| RaceId | RaceResults |
| --- | --- |
| 1 | ["racer3","racer5","racer2","racer1","racer6"] |
| 2 | ["racer6","racer4","racer2","racer1","racer3","racer5"] |
| 3 | ["racer4","racer3","racer5","racer2","racer6","racer1"] |
| 4 | ["racer1","racer2","racer3","racer5"] |
| 5 | ["racer5","racer2","racer4","racer6","racer3"] |

**Transformation:**

Note that the number of racers varies with each race, so determining the position of the last racer depends on the number in the event. The number of racers can be captured using the following:

| Transformation Name | New formula |
| --- | --- |
| **Parameter: Formula type** | Single row formula |
| **Parameter: Formula** | ARRAYLEN(RaceResults) |
| **Parameter: New column name** | 'countRacers' |

Create columns containing the index values for each racer. Below is the example for `racer1`:

| Transformation Name | New formula |
| --- | --- |
| **Parameter: Formula type** | Single row formula |
| **Parameter: Formula** | ARRAYINDEXOF(RaceResults, 'racer1') |
| **Parameter: New column name** | 'arrL-IndexRacer1' |

| Transformation Name | New formula |
| --- | --- |
| **Parameter: Formula type** | Single row formula |
| **Parameter: Formula** | ARRAYRIGHTINDEXOF(RaceResults, 'racer1') |
| **Parameter: New column name** | 'arrR-IndexRacer1' |

You can then compare the values in the two columns to determine if they are the same.

> **NOTE:** If ARRAYINDEXOF and ARRAYRIGHTINDEXOF do not return the same value for the same inputs, then the value is not unique in the array.

Since the points awarded for 1st, 2nd, and 3rd place follow a consistent pattern, you can use the following single statement to compute points for podium finishes for `racer1`: computing based on the value stored for the left index value:

| Transformation Name | Conditional column |
|---|---|
| Parameter: Condition type | if...then...else |
| Parameter: If | {arrayL-IndexRacer1} < 3 |
| Parameter: Then | (3 - {arrayL-IndexRacer1}) * 10 |
| Parameter: Else | 0 |
| Parameter: New column name | 'ptsRacer1' |

The following transform then edits the `ptsRacer1` to evaluate for the Did Not Finish (DNF) and last place conditions:

| Transformation Name | Edit column with formula |
|---|---|
| Parameter: Columns | ptsRacer1 |
| Parameter: Formula | IF(ISNULL({arrayL-IndexRacer1}), -20, ptsRacer1)) |

You can use the following to determine if the specified racer was last in the event:

| Transformation Name | Edit column with formula |
|---|---|
| Parameter: Columns | ptsRacer1 |
| Parameter: Formula | IF(arrR-IndexRacer1 == countRacers, -10, ptsRacer1) |

**Results:**

| RaceId | RaceResults | countRacers | arrR-IndexRacer1 | arrL-IndexRacer1 | ptsRacer1 |
|---|---|---|---|---|---|
| 1 | ["racer3","racer5","racer2","racer1","racer6"] | 5 | 3 | 3 | 0 |
| 2 | ["racer6","racer4","racer2","racer1","racer3"," racer5"] | 6 | 3 | 3 | 0 |
| 3 | ["racer4","racer3","racer5","racer2","racer6"," racer1"] | 6 | 5 | 5 | -10 |
| 4 | ["racer1","racer2","racer3","racer5"] | 4 | 0 | 0 | 20 |
| 5 | ["racer5","racer2","racer4","racer6","racer3"] | 5 | null | null | -20 |

# ARRAYINTERSECT Function

Generates an array containing all elements that appear in multiple input arrays, referenced as column names or array literals.

**Wrangle vs. SQL:** This function is part of Wrangle , a proprietary data transformation language. Wrangle is not SQL. For more information, see *Wrangle Language*.

## Basic Usage

**Array literal reference example:**

```
arrayintersect([["A","B","C"],["A","D",
"E"]])
```

**Output:** Returns a single array with a single element:

```
["A"]
```

**Column reference example:**

```
arrayintersect([array1,array2])
```

**Output:** Returns a single array listing all of the elements that appear in both `array1` and `array2` in order.

## Syntax and Arguments

```
arrayintersect(array_ref1,array_ref2)
```

| Argument | Required? | Data Type | Description |
|----------|-----------|-----------|-------------|
| array_ref1 | Y | string or array | Name of first column or first array literal to apply to the function |
| array_ref2 | Y | string or array | Name of second column or second array literal to apply to the function |

For more information on syntax standards, see *Language Documentation Syntax Notes*.

**array_ref1, array_ref2**

Array literal or name of the array column whose intersection you want to derive. You can intersect two or more array columns together.

**Usage Notes:**

| Required? | Data Type | Example Value |
|-----------|-----------|---------------|
| Yes | Array literal or column reference | `myArray1, myArray2` |

## Examples

> **Tip:** For additional examples, see *Common Tasks*.

## Example - Simple intersection example

This example illustrates how to use the nested functions.

**Functions**:

| Item | Description |
|------|-------------|
| ARRAYCONCAT Function | Combines the elements of one array with another, listing all elements of the first array before listing all elements of the second array. |
| ARRAYINTERSECT Function | Generates an array containing all elements that appear in multiple input arrays, referenced as column names or array literals. |
| ARRAYCROSS Function | Generates a nested array containing the cross-product of all elements in two or more arrays. |
| ARRAYUNIQUE Function | Generates an array of all unique elements among one or more arrays. |

**Source:**

Code formatting has been applied to improve legibility.

| Item | ArrayA | ArrayB |
|------|--------|--------|
| Item1 | `["A","B","C"]` | `["1","2","3"]` |
| Item2 | `["A","B"]` | `["A","B","C"]` |
| Item3 | `["D","E","F"]` | `["4","5","6"]` |

**Transformation:**

You can apply the following transforms in the following order. Note that the column names must be different from the transform name, which is a reserved word.

| Transformation Name | `New formula` |
|---------------------|---------------|
| **Parameter: Formula type** | `Single row formula` |
| **Parameter: Formula** | `ARRAYCONCAT([Letters,Numerals])` |
| **Parameter: New column name** | `'concat2'` |

| Transformation Name | `New formula` |
|---------------------|---------------|
| **Parameter: Formula type** | `Single row formula` |
| **Parameter: Formula** | `ARRAYINTERSECT([Letters,Numerals])` |
| **Parameter: New column name** | `'intersection2'` |

| Transformation Name | `New formula` |
|---------------------|---------------|
| **Parameter: Formula type** | `Single row formula` |
|  |  |

| Parameter: Formula | ARRAYCROSS([Letters,Numerals]) |
|---|---|
| Parameter: New column name | 'cross2' |

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | ARRAYUNIQUE([Letters,Numerals]) |
| Parameter: New column name | 'unique2' |

**Results:**

For display purposes, the results table has been broken down into three separate sets of columns.

Column set 1:

| Item | ArrayA | ArrayB | concat2 | intersection2 |
|---|---|---|---|---|
| Item1 | ["A","B","C"] | ["1","2","3"] | ["A","B","C","1","2","3"] | [ ] |
| Item2 | ["A","B"] | ["A","B","C"] | ["A","B","A","B","C"] | ["A","B"] |
| Item3 | ["D","E","F"] | ["4","5","6"] | ["D","E","F","4","5","6"] | [ ] |

Column set 2:

| Item | cross2 |
|---|---|
| Item1 | [["A","1"],["A","2"],["A","3"],["B","1"],["B","2"],["B","3"],["C","1"],["C","2"],["C","3"]] |
| Item2 | [["A","A"],["A","B"],["A","C"],["B","A"],["B","B"],["B","C"]] |
| Item3 | [["D","4"],["D","5"],["D","6"],["E","4"],["E","5"],["E","6"], ["F","4"], ["F","5"],["F","6"]] |

Column set 3:

| Item | unique2 |
|---|---|
| Item1 | ["A","B","C","1","2","3"] |
| Item2 | ["A","B","C"] |
| Item3 | ["D","E","F","4","5","6"] |

# ARRAYLEN Function

Computes the number of elements in the arrays in the specified column, array literal, or function that returns an array.

- This function calculates the number of elements in the outer layer of an array. If your array is nested, the count of inner elements is not factored.
- If a row contains a missing array, the returned value is `0`. If it contains a value that is not recognized as an array, the returned value is blank.

**Wrangle vs. SQL:** This function is part of Wrangle , a proprietary data transformation language. Wrangle is not SQL. For more information, see *Wrangle Language*.

## Basic Usage

**Array literal reference example:**

```
arraylen([A,B,C,D])
```

**Output:** Returns the count of elements in the array, which is `4`.

**Column reference example:**

```
arraylen([myValues])
```

**Output:** Returns the count of elements in the `myValues` column.

**Array function example:**

```
arraylen(concat([colA,colB]))
```

**Output:** Returns the count of elements in the array returned from concatenating `colA` and `colB`.

## Syntax and Arguments

```
arraylen(array_ref)
```

| Argument | Required? | Data Type | Description |
|----------|-----------|-----------|-------------|
| array_ref | Y | string | Name of Array column, Array literal, or function returning an Array to apply to the function |

For more information on syntax standards, see *Language Documentation Syntax Notes*.

**array_ref**

Name of the array column, array literal, or function returning an array whose elements you want to count.

- Multiple columns and wildcards are not supported.

**Usage Notes:**

| Required? | Data Type | Example Value |
|---|---|---|
| Yes | String (column reference or function) or array literal | `myArray1` |

## Examples

> **Tip:** For additional examples, see *Common Tasks*.

### Example - Unnest an array

This example illustrates you to use the flatten and unnest transforms.

**Source:**

You have the following data on student test scores. Scores on individual scores are stored in the `Scores` array, and you need to be able to track each test on a uniquely identifiable row. This example has two goals:

1. One row for each student test
2. Unique identifier for each student-score combination

| LastName | FirstName | Scores |
|---|---|---|
| Adams | Allen | [81,87,83,79] |
| Burns | Bonnie | [98,94,92,85] |
| Cannon | Charles | [88,81,85,78] |

**Transformation:**

When the data is imported from CSV format, you must add a `header` transform and remove the quotes from the `Scores` column:

| Transformation Name | `Rename column with row(s)` |
|---|---|
| **Parameter: Option** | `Use row(s) as column names` |
| **Parameter: Type** | `Use a single row to name columns` |
| **Parameter: Row number** | `1` |

| Transformation Name | `Replace text or pattern` |
|---|---|
| **Parameter: Column** | `colScores` |
| **Parameter: Find** | `'\"'` |
| **Parameter: Replace with** | `''` |
| **Parameter: Match all occurrences** | `true` |

**Validate test date:** To begin, you might want to check to see if you have the proper number of test scores for each student. You can use the following transform to calculate the difference between the expected number of elements in the `Scores` array (4) and the actual number:

| | |
|---|---|
| Transformation Name | New formula |
| Parameter: Formula type | Single row formula |
| Parameter: Formula | (4 - arraylen(Scores)) |
| Parameter: New column name | 'numMissingTests' |

When the transform is previewed, you can see in the sample dataset that all tests are included. You might or might not want to include this column in the final dataset, as you might identify missing tests when the recipe is run at scale.

**Unique row identifier:** The Scores array must be broken out into individual rows for each test. However, there is no unique identifier for the row to track individual tests. In theory, you could use the combination of LastName-FirstName-Scores values to do so, but if a student recorded the same score twice, your dataset has duplicate rows. In the following transform, you create a parallel array called Tests, which contains an index array for the number of values in the Scores column. Index values start at 0:

| | |
|---|---|
| Transformation Name | New formula |
| Parameter: Formula type | Single row formula |
| Parameter: Formula | range(0,arraylen(Scores)) |
| Parameter: New column name | 'Tests' |

Also, we will want to create an identifier for the source row using the sourcerownumber function:

| | |
|---|---|
| Transformation Name | New formula |
| Parameter: Formula type | Single row formula |
| Parameter: Formula | sourcerownumber() |
| Parameter: New column name | 'orderIndex' |

**One row for each student test:** Your data should look like the following:

| LastName | FirstName | Scores | Tests | orderIndex |
|---|---|---|---|---|
| Adams | Allen | [81,87,83,79] | [0,1,2,3] | 2 |
| Burns | Bonnie | [98,94,92,85] | [0,1,2,3] | 3 |
| Cannon | Charles | [88,81,85,78] | [0,1,2,3] | 4 |

Now, you want to bring together the Tests and Scores arrays into a single nested array using the arrayzip function:

| | |
|---|---|
| Transformation Name | New formula |
| Parameter: Formula type | Single row formula |
| Parameter: Formula | arrayzip([Tests,Scores]) |

Your dataset has been changed:

| LastName | FirstName | Scores | Tests | orderIndex | column1 |
|---|---|---|---|---|---|
| Adams | Allen | [81,87,83,79] | [0,1,2,3] | 2 | [[0,81],[1,87],[2,83],[3,79]] |
| Adams | Bonnie | [98,94,92,85] | [0,1,2,3] | 3 | [[0,98],[1,94],[2,92],[3,85]] |
| Cannon | Charles | [88,81,85,78] | [0,1,2,3] | 4 | [[0,88],[1,81],[2,85],[3,78]] |

Use the following to unpack the nested array:

| Transformation Name | Expand arrays to rows |
|---|---|
| Parameter: Column | column1 |

Each test-score combination is now broken out into a separate row. The nested Test-Score combinations must be broken out into separate columns using the following:

| Transformation Name | Unnest Objects into columns |
|---|---|
| Parameter: Column | column1 |
| Parameter: Paths to elements | '[0]','[1]' |

After you delete `column1`, which is no longer needed you should rename the two generated columns:

| Transformation Name | Rename columns |
|---|---|
| Parameter: Option | Manual rename |
| Parameter: Column | column_0 |
| Parameter: New column name | 'TestNum' |

| Transformation Name | Rename columns |
|---|---|
| Parameter: Option | Manual rename |
| Parameter: Column | column_1 |
| Parameter: New column name | 'TestScore' |

**Unique row identifier:** You can do one more step to create unique test identifiers, which identify the specific test for each student. The following uses the original row identifier `OrderIndex` as an identifier for the student and the `TestNumber` value to create the `TestId` column value:

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | (orderIndex * 10) + TestNum |
| Parameter: New column name | 'TestId' |

The above are integer values. To make your identifiers look prettier, you might add the following:

| Transformation Name | Merge columns |
|---|---|
| | |

| Parameter: Columns | 'TestId00','TestId' |
|---|---|

**Extending:** You might want to generate some summary statistical information on this dataset. For example, you might be interested in calculating each student's average test score. This step requires figuring out how to properly group the test values. In this case, you cannot group by the `LastName` value, and when executed at scale, there might be collisions between first names when this recipe is run at scale. So, you might need to create a kind of primary key using the following:

| Transformation Name | Merge columns |
|---|---|
| Parameter: Columns | 'LastName','FirstName' |
| Parameter: Separator | '-' |
| Parameter: New column name | 'studentId' |

You can now use this as a grouping parameter for your calculation:

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | average(TestScore) |
| Parameter: Group rows by | studentId |
| Parameter: New column name | 'avg_TestScore' |

**Results:**

After you delete unnecessary columns and move your columns around, the dataset should look like the following:

| TestId | LastName | FirstName | TestNum | TestScore | studentId | avg_TestScore |
|---|---|---|---|---|---|---|
| TestId0021 | Adams | Allen | 0 | 81 | Adams-Allen | 82.5 |
| TestId0022 | Adams | Allen | 1 | 87 | Adams-Allen | 82.5 |
| TestId0023 | Adams | Allen | 2 | 83 | Adams-Allen | 82.5 |
| TestId0024 | Adams | Allen | 3 | 79 | Adams-Allen | 82.5 |
| TestId0031 | Adams | Bonnie | 0 | 98 | Adams-Bonnie | 92.25 |
| TestId0032 | Adams | Bonnie | 1 | 94 | Adams-Bonnie | 92.25 |
| TestId0033 | Adams | Bonnie | 2 | 92 | Adams-Bonnie | 92.25 |
| TestId0034 | Adams | Bonnie | 3 | 85 | Adams-Bonnie | 92.25 |
| TestId0041 | Cannon | Chris | 0 | 88 | Cannon-Chris | 83 |
| TestId0042 | Cannon | Chris | 1 | 81 | Cannon-Chris | 83 |
| TestId0043 | Cannon | Chris | 2 | 85 | Cannon-Chris | 83 |
| TestId0044 | Cannon | Chris | 3 | 78 | Cannon-Chris | 83 |

# ARRAYMERGEELEMENTS Function

**Contents:**

---

Merges the elements of an array in left to right order into a string. Values are optionally delimited by a provided delimiter.

**Wrangle vs. SQL:** This function is part of Wrangle , a proprietary data transformation language. Wrangle is not SQL. For more information, see *Wrangle Language*.

## Basic Usage

**Array literal reference example:**

```
arraymergeelements([&quot;A&quot;,&quot;B&quot;,&quot;C&quot;,&quot;D&quot;],&quot;-&quot;)
```

**Output:** Returns the following String value: `"A-B-C-D"`.

**Column reference example:**

```
arraymergeelements([myValues)
```

**Output:** Generates the new `myValuesMergedTogether` column containing all of the elements in the arrays in the `myElement` column joined together without a delimiter between them.

## Syntax and Arguments

```
arraymergeelements(array_ref,my_element, [string_delimiter])
```

| Argument | Required? | Data Type | Description |
|---|---|---|---|
| array_ref | Y | array | Name of Array column, Array literal, or function returning an Array to apply to the function |
| string_delimiter | Y | string | Optional String delimiter to insert between merged elements in the output String. |

For more information on syntax standards, see *Language Documentation Syntax Notes*.

**array_ref**

Name of the array column, array literal, or function whose elements you wish to merge.

- Multiple columns and wildcards are not supported.

**Usage Notes:**

| Required? | Data Type | Example Value |
|-----------|-----------|---------------|
| Yes | String (column reference or function) or array literal | `myArray1` |

### string_delimiter

Optional string value to insert between elements in the merged output string.

**Usage Notes:**

| Required? | Data Type | Example Value |
|-----------|-----------|---------------|
| No | String | `" - "` |

## Examples

> **Tip:** For additional examples, see *Common Tasks*.

### Example - Podium Race Finishes

This example illustrates how to generate an Array that is a slice of an another Array, based on index numbers. The elements of this Array can then be merged into a String value.

**Functions**:

| Item | Description |
|------|-------------|
| ARRAYSLICE Function | Returns an array containing a slice of the input array, as determined by starting and ending index parameters. |
| ARRAYMERGEELEMENTS Function | Merges the elements of an array in left to right order into a string. Values are optionally delimited by a provided delimiter. |

**Source:**

The following set of arrays contain results, in order, of a series of races. From this list, the goal is to extract a list of the podium finishers for each race as a single string.

| RaceId | RaceResults |
|--------|-------------|
| 1 | ["racer3","racer5","racer2","racer1","racer6"] |
| 2 | ["racer6","racer4","racer2","racer1","racer3","racer5"] |
| 3 | ["racer4","racer3","racer5","racer2","racer6","racer1"] |
| 4 | ["racer1","racer2","racer3","racer5"] |
| 5 | ["racer5","racer2","racer4","racer6","racer3"] |

**Transformation:**

From the list of arrays, the first step is to gather the top-3 finishers from each race:

| Transformation Name | `New formula` |
|---|---|
| **Parameter: Formula type** | `Single row formula` |
| **Parameter: Formula** | `ARRAYSLICE(RaceResults, 0, 3)` |
| **Parameter: New column name** | `'arrPodium'` |

The above captures the first three values of the RaceResults arrays into a new set of arrays.

The next step is to merge this new set of arrays into a single string:

| Transformation Name | `New formula` |
|---|---|
| **Parameter: Formula type** | `Single row formula` |
| **Parameter: Formula** | `ARRAYMERGEELEMENTS(arrPodium, ',')` |
| **Parameter: New column name** | `'strPodium'` |

**Results:**

| RaceId | RaceResults | arrPodium | strPodium |
|---|---|---|---|
| 1 | ["racer3","racer5","racer2","racer1","racer6"] | ["racer3","racer5","racer2"] | racer3,racer5,racer2 |
| 2 | ["racer6","racer4","racer2","racer1","racer3","racer5"] | ["racer6","racer4","racer2"] | racer6,racer4,racer2 |
| 3 | ["racer4","racer3","racer5","racer2","racer6","racer1"] | ["racer4","racer3","racer5"] | racer4,racer3,racer5 |
| 4 | ["racer1","racer2","racer3","racer5"] | ["racer1","racer2","racer3"] | racer1,racer2,racer3 |
| 5 | ["racer5","racer2","racer4","racer6","racer3"] | ["racer5","racer2","racer4"] | racer5,racer2,racer4 |

# ARRAYRIGHTINDEXOF Function

**Contents:**

---

Computes the index at which a specified element is first found within an array, when searching right to left. Returned value is based on left-to-right indexing.

- Leftmost index value is 0. Rightmost index value is the same value returned by ARRAYLEN function.
- If the element is not found, null is returned.
- For left-to-right searching, use ARRAYINDEXOF.
    - If only one element exists in the array, both functions return the same value.
    - For more information, see *ARRAYINDEXOF Function*.

**Wrangle vs. SQL:** This function is part of Wrangle , a proprietary data transformation language. Wrangle is not SQL. For more information, see *Wrangle Language*.

## Basic Usage

**Array literal reference example:**

```
arrayrightindexof(["A","B","C","D"],"C")
```

**Output:** Returns the right-index of the element "C" in the array, which is 2 in an 0-based index from left to right.

**Column reference example:**

```
arrayrightindexof([myValues],myElement)
```

**Output:** Returns the right-search in the `myValues` arrays for the elements listed in the `myElement` column.

## Syntax and Arguments

```
arrayrightindexof(array_ref,my_element)
```

| Argument | Required? | Data Type | Description |
|----------|-----------|-----------|-------------|
| array_ref | Y | array or string | Name of Array column, Array literal, or function returning an Array to apply to the function |
| my_element | Y | any | The element to locate in the array, searching from right to left |

For more information on syntax standards, see *Language Documentation Syntax Notes*.

### array_ref

Name of the array column, array literal, or function returning an array whose element you want to locate.

- Multiple columns and wildcards are not supported.

**Usage Notes:**

| Required? | Data Type | Example Value |
|---|---|---|
| Yes | String (column reference or function) or array literal | `myArray1` |

### my_element

Element literal that you wish to locate in the array. It can be a value of any data type.

**Usage Notes:**

| Required? | Data Type | Example Value |
|---|---|---|
| Yes | Any | `"1st"` |

## Examples

> **Tip:** For additional examples, see *Common Tasks*.

### Example - Computing points based on position of finish

This example illustrates how to convert the index value of an array for a specified value searching from left to right and right to left by using ARRAYINDEXOF and ARRAYRIGTHINDEXOF functions.

**Functions**:

| Item | Description |
|---|---|
| ARRAYINDEXOF Function | Computes the index at which a specified element is first found within an array. Indexing is left to right. |
| ARRAYRIGHTINDEXOF Function | Computes the index at which a specified element is first found within an array, when searching right to left. Returned value is based on left-to-right indexing. |

**Source:**

The following set of arrays contain results, in order, of a series of races. From this list, the goal is to generate the score for each racer according to the following scoring matrix.

| Place | Points |
|---|---|
| 1st | 30 |
| 2nd | 20 |
| 3rd | 10 |

| Last | -10 |
|---|---|
| Did Not Finish (DNF) | -20 |

Results:

| RaceId | RaceResults |
|---|---|
| 1 | ["racer3","racer5","racer2","racer1","racer6"] |
| 2 | ["racer6","racer4","racer2","racer1","racer3","racer5"] |
| 3 | ["racer4","racer3","racer5","racer2","racer6","racer1"] |
| 4 | ["racer1","racer2","racer3","racer5"] |
| 5 | ["racer5","racer2","racer4","racer6","racer3"] |

**Transformation:**

Note that the number of racers varies with each race, so determining the position of the last racer depends on the number in the event. The number of racers can be captured using the following:

| Transformation Name | New formula |
|---|---|
| **Parameter: Formula type** | Single row formula |
| **Parameter: Formula** | ARRAYLEN(RaceResults) |
| **Parameter: New column name** | 'countRacers' |

Create columns containing the index values for each racer. Below is the example for `racer1`:

| Transformation Name | New formula |
|---|---|
| **Parameter: Formula type** | Single row formula |
| **Parameter: Formula** | ARRAYINDEXOF(RaceResults, 'racer1') |
| **Parameter: New column name** | 'arrL-IndexRacer1' |

| Transformation Name | New formula |
|---|---|
| **Parameter: Formula type** | Single row formula |
| **Parameter: Formula** | ARRAYRIGHTINDEXOF(RaceResults, 'racer1') |
| **Parameter: New column name** | 'arrR-IndexRacer1' |

You can then compare the values in the two columns to determine if they are the same.

> **NOTE:** If ARRAYINDEXOF and ARRAYRIGHTINDEXOF do not return the same value for the same inputs, then the value is not unique in the array.

Since the points awarded for 1st, 2nd, and 3rd place follow a consistent pattern, you can use the following single statement to compute points for podium finishes for `racer1`: computing based on the value stored for the left index value:

| Transformation Name | Conditional column |
|---|---|
| Parameter: Condition type | if...then...else |
| Parameter: If | {arrayL-IndexRacer1} < 3 |
| Parameter: Then | (3 - {arrayL-IndexRacer1}) * 10 |
| Parameter: Else | 0 |
| Parameter: New column name | 'ptsRacer1' |

The following transform then edits the `ptsRacer1` to evaluate for the Did Not Finish (DNF) and last place conditions:

| Transformation Name | Edit column with formula |
|---|---|
| Parameter: Columns | ptsRacer1 |
| Parameter: Formula | IF(ISNULL({arrayL-IndexRacer1}), -20, ptsRacer1)) |

You can use the following to determine if the specified racer was last in the event:

| Transformation Name | Edit column with formula |
|---|---|
| Parameter: Columns | ptsRacer1 |
| Parameter: Formula | IF(arrR-IndexRacer1 == countRacers, -10, ptsRacer1) |

**Results:**

| RaceId | RaceResults | countRacers | arrR-IndexRacer1 | arrL-IndexRacer1 | ptsRacer1 |
|---|---|---|---|---|---|
| 1 | ["racer3","racer5","racer2","racer1","racer6"] | 5 | 3 | 3 | 0 |
| 2 | ["racer6","racer4","racer2","racer1","racer3","racer5"] | 6 | 3 | 3 | 0 |
| 3 | ["racer4","racer3","racer5","racer2","racer6","racer1"] | 6 | 5 | 5 | -10 |
| 4 | ["racer1","racer2","racer3","racer5"] | 4 | 0 | 0 | 20 |
| 5 | ["racer5","racer2","racer4","racer6","racer3"] | 5 | null | null | -20 |

# ARRAYSLICE Function

**Contents:**

Returns an array containing a slice of the input array, as determined by starting and ending index parameters.

- Starting index parameter is required. A value of `0` indicates the first element of the array.
- Ending index parameter is optional.
  - Ending index value is 0-based and not inclusive.
  - Default value is empty, which indicates the end of the array.

**Wrangle vs. SQL:** This function is part of Wrangle , a proprietary data transformation language. Wrangle is not SQL. For more information, see *Wrangle Language*.

## Basic Usage

**Array literal reference example:**

```
arrayslice(["A","B","C","D"],1,2)
```

**Output:** Returns the array: `["B"]`.

**Column reference example:**

```
arrayslice([myValues],2)
```

**Output:** Returns a slice of the arrays in the `myValues` column, starting at the third value and extending to the end of the array.

## Syntax and Arguments

```
arrayslice(array_ref,int_start_index,[int_end_index])
```

| Argument | Required? | Data Type | Description |
|---|---|---|---|
| array_ref | Y | array or string | Name of Array column, Array literal, or function returning an Array to apply to the function |
| int_start_index | Y | integer | 0-based index value of the first element in the Array to include in the slice. |
| int_end_index | N | integer | 0-based index "soft" value of the last element in the Array to include in the slice. Listed value is not included.<br><br>If no value is provided, the last element of the array is the end of the slice. |

For more information on syntax standards, see *Language Documentation Syntax Notes*.

**array_ref**

Name of the array column, array literal, or function returning an array whose element you want to locate.

- Multiple columns and wildcards are not supported.

**Usage Notes:**

| Required? | Data Type | Example Value |
|-----------|-----------|---------------|
| Yes | String (column reference or function) or array literal | `myArray1` |

**int_start_index**

Index of the starting element of the source array that you wish to include in the slice.

- A value of `0` captures the first element of the array.
- If this value is greater than the total number of elements in the source array, an empty array is returned as the slice.

**Usage Notes:**

| Required? | Data Type | Example Value |
|-----------|-----------|---------------|
| Yes | Integer (whole number) | `4` |

**int_end_index**

Optional index of the ending element of the source array that you wish to include in the slice.

- A value of `0` captures the first element of the array.
- The value indicated by this parameter is not included in the slice.
- If the end index value is specified, it must be greater than or equal to the start index value.
- If this value is greater than the total number of elements in the source array, then the slice ends at the final element of the array.

**Usage Notes:**

| Required? | Data Type | Example Value |
|-----------|-----------|---------------|
| Yes | Integer (whole number) | `10` |

## Examples

> **Tip:** For additional examples, see *Common Tasks*.

### Example - Podium Race Finishes

This example illustrates how to generate an Array that is a slice of an another Array, based on index numbers. The elements of this Array can then be merged into a String value.

**Functions**:

| Item | Description |
|------|-------------|
| ARRAYSLICE Function | Returns an array containing a slice of the input array, as determined by starting and ending index parameters. |
| ARRAYMERGEELEMENTS Function | Merges the elements of an array in left to right order into a string. Values are optionally delimited by a provided delimiter. |

### Source:

The following set of arrays contain results, in order, of a series of races. From this list, the goal is to extract a list of the podium finishers for each race as a single string.

| RaceId | RaceResults |
|--------|-------------|
| 1 | ["racer3","racer5","racer2","racer1","racer6"] |
| 2 | ["racer6","racer4","racer2","racer1","racer3","racer5"] |
| 3 | ["racer4","racer3","racer5","racer2","racer6","racer1"] |
| 4 | ["racer1","racer2","racer3","racer5"] |
| 5 | ["racer5","racer2","racer4","racer6","racer3"] |

### Transformation:

From the list of arrays, the first step is to gather the top-3 finishers from each race:

| Transformation Name | New formula |
|---------------------|-------------|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | ARRAYSLICE(RaceResults, 0, 3) |
| Parameter: New column name | 'arrPodium' |

The above captures the first three values of the RaceResults arrays into a new set of arrays.

The next step is to merge this new set of arrays into a single string:

| Transformation Name | New formula |
|---------------------|-------------|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | ARRAYMERGEELEMENTS(arrPodium, ',') |
| Parameter: New column name | 'strPodium' |

### Results:

| RaceId | RaceResults | arrPodium | strPodium |
|---|---|---|---|
| 1 | ["racer3","racer5","racer2","racer1","racer6"] | ["racer3","racer5","racer2"] | racer3,racer5,racer2 |
| 2 | ["racer6","racer4","racer2","racer1","racer3","racer5"] | ["racer6","racer4","racer2"] | racer6,racer4,racer2 |
| 3 | ["racer4","racer3","racer5","racer2","racer6","racer1"] | ["racer4","racer3","racer5"] | racer4,racer3,racer5 |
| 4 | ["racer1","racer2","racer3","racer5"] | ["racer1","racer2","racer3"] | racer1,racer2,racer3 |
| 5 | ["racer5","racer2","racer4","racer6","racer3"] | ["racer5","racer2","racer4"] | racer5,racer2,racer4 |

# ARRAYSORT Function

**Contents:**

Sorts array values in the specified column, array literal, or function that returns an array in ascending or descending order.

- This function calculates based on the outer layer of an array. If your array is nested, the sorting of inner elements is not factored.

**Wrangle vs. SQL:** This function is part of Wrangle , a proprietary data transformation language. Wrangle is not SQL. For more information, see *Wrangle Language*.

## Basic Usage

**Array literal reference example:**

```
arraysort([A,B,C,D],descending)
```

**Output:** Returns the following array: `[D,C,B,A]`.

**Column reference example:**

```
arraysort(myArrays,ascending)
```

**Output:** Returns the arrays listed in the `myArrays` column sorted in ascending order.

## Syntax and Arguments

```
arraysort(array_ref,order_enum)
```

| Argument | Required? | Data Type | Description |
|----------|-----------|-----------|-------------|
| array_ref | Y | string | Name of Array column, Array literal, or function returning an Array to apply to the function |
| order_enum | Y | string (enumerated value) | Order is defined as either:<br><br>• `ascending` (default)<br>• `descending` |

For more information on syntax standards, see *Language Documentation Syntax Notes*.

**array_ref**

Name of the array column, array literal, or function returning an array whose array values you wish to sort.

- Multiple columns and wildcards are not supported.

**Usage Notes:**

| Required? | Data Type | Example Value |
|---|---|---|
| Yes | String (column reference or function) or array literal | `myArray1` |

**order_enum**

String literal indicating the order by which the referenced arrays should be sorted:

- `ascending` - (default) lowest values for the valid data type are listed first.
- `descending` - Null/empty values are sorted first, followed by mismatched values. Then, the array values that are valid for the specified data type are listed in descending order.

**Usage Notes:**

| Required? | Data Type | Example Value |
|---|---|---|
| No | String (enumerated value) | `descending` |

## Examples

> **Tip:** For additional examples, see *Common Tasks*.

**Example - Student progress across tests**

This example illustrates how to return n-based number of elements in an array.

**Functions**:

| Item | Description |
|---|---|
| ARRAYLEN Function | Computes the number of elements in the arrays in the specified column, array literal, or function that returns an array. |
| ARRAYELEMENTAT Function | Computes the 0-based index value for an array element in the specified column, array literal, or function that returns an array. |
| ARRAYSORT Function | Sorts array values in the specified column, array literal, or function that returns an array in ascending or descending order. |

**Source:**

Here are some student test scores. Individual scores are stored in the `Scores` column. You want to:

1. Flag the students who have not taken four tests.

2. Compute the range in scores for each student.

| LastName | FirstName | Scores |
|----------|-----------|--------|
| Allen | Amanda | [79, 83,87,81] |
| Bell | Bobby | [85, 92, 94, 98] |
| Charles | Cameron | [88,81,85] |
| Dudley | Danny | [82,88,81,77] |
| Ellis | Evan | [91,93,87,93] |

**Transformation:**

First, you want to flag the students who did not take all four tests:

| Transformation Name | New formula |
|---------------------|-------------|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | IF(ARRAYLEN(Scores) < 4,"incomplete","") |
| Parameter: New column name | 'Error' |

This test flags Cameron Charles only.

The following transform sorts the array values in highest to lowest score:

| Transformation Name | Edit column with formula |
|---------------------|--------------------------|
| Parameter: Columns | Scores |
| Parameter: Formula | ARRAYSORT(Scores, 'descending') |

The following transforms extracts the first (highest) and last (lowest) value in each student's test scores, provided that they took four tests:

| Transformation Name | New formula |
|---------------------|-------------|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | ARRAYELEMENTAT(Scores,0) |
| Parameter: New column name | 'highestScore' |

| Transformation Name | New formula |
|---------------------|-------------|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | ARRAYELEMENTAT(Scores,3) |
| Parameter: New column name | 'lowestScore' |

> **Tip:** You could also generate the `Error` column when the `Scores4` column contains a null value. If no value exists in the array for the `ARRAYELEMENTAT` function, a null value is returned, which would indicate in this case an insufficient number of elements (test scores).

You can now track change in test scores:

| Transformation Name | New formula |
|---|---|
| **Parameter: Formula type** | `Single row formula` |
| **Parameter: Formula** | `SUBTRACT(highestScore,lowestScore)` |
| **Parameter: New column name** | `'Score_range'` |

**Results:**

| LastName | FirstName | Scores | Error | lowestScore | highestScore | Score_range |
|---|---|---|---|---|---|---|
| Allen | Amanda | [87,83,81,79] | | 79 | 87 | 8 |
| Bell | Bobby | [98,94,92,85] | | 85 | 98 | 13 |
| Charles | Cameron | [88,85,81] | incomplete | | 88 | |
| Dudley | Danny | [88,82,81,77] | | 77 | 88 | 11 |
| Ellis | Evan | [93,93,91,87] | | 87 | 93 | 6 |

# ARRAYSTOMAP Function

**Contents:**

Combines one array containing keys and another array containing values into an Object of key-value pairs.

- This function applies to two inputs only.
- Inputs can be array literals, column references, or functions returning arrays.

If the number of key elements is greater than the number of value elements, null values are generated for the missing values in the output Object. If the number of value elements is greater, the `DEFAULT_KEY` value (third parameter) is applied.

**Wrangle vs. SQL:** This function is part of Wrangle , a proprietary data transformation language. Wrangle is not SQL. For more information, see *Wrangle Language*.

## Basic Usage

**Array literal reference example:**

```
arraystomap(["A","B"],["1","2","3"])
```

**Output:** Returns an Object associating keys from the first array with values from the second array.

**Column reference example:**

```
arraystomap(array1,array2, 'extraProps')
```

**Output:** Returns an Object pairing the elements of the arrays as key-value pairs. Any extra values in `array2` are assigned to the `extraProps` key.

**Function reference example:**

```
arraystomap(array1,concat([array2,array3]))
```

**Output:** Returns an Object pairing the elements of `array1` and the array created by concatenating `array2` and `array3`.

## Syntax and Arguments

```
arraystomap(array_keys,array_values, ['DEFAULT KEY'])
```

| Argument | Required? | Data | Description |
|---|---|---|---|

| | | Type | |
|---|---|---|---|
| array_keys | Y | string or array | Name of column, array literal, or function returning an array whose elements are the keys for the generated Object |
| array_values | Y | string or array | Name of column, array literal, or function returning an array whose elements are the values for the generated Object |
| DEFAULT_ KEY | N | string literal | Any extra values are assigned to this specified key |

For more information on syntax standards, see *Language Documentation Syntax Notes*.

### array_keys

Name of the array column, array literal, or function returning an array whose elements you want to use as the keys for the Object.

**Usage Notes:**

| Required? | Data Type | Example Value |
|---|---|---|
| Yes | String (column reference, function returning an array) or array literal | `myKeys` |

### array_values

Name of the array column, array literal, or function returning an array whose elements you want to use as the values in the Object.

**Usage Notes:**

| Required? | Data Type | Example Value |
|---|---|---|
| Yes | String (column reference, function returning an array) or array literal | `myValues` |

### DEFAULT KEY

If there are extra elements in the second array, they are assigned to the key that is defined by this parameter.

**Usage Notes:**

| Required? | Data Type | Example Value |
|---|---|---|
| No | String literal | `'extraProperties'` |

## Examples

> **Tip:** For additional examples, see *Common Tasks*.

### Example - Create an Object of product properties

This example illustrates how to use the ARRAYSTOMAP and KEYS functions to convert values in Array or Object data type of key-value pairs.

**Functions:**

| | |
|---|---|
| | |

| Item | Description |
|---|---|
| ARRAYSTOMAP Function | Combines one array containing keys and another array containing values into an Object of key-value pairs. |
| KEYS Function | Extracts the key values from an Object data type column and stores them in an array of String values. |

**Source:**

Your dataset contains master product data with product properties stored in two arrays of keys and values.

| ProdId | ProdCategory | ProdName | ProdKeys | ProdProperties |
|---|---|---|---|---|
| S001 | Shirts | Crew Neck T-Shirt | ["type", "color", "fabric", "sizes"] | ["crew","blue","cotton","S,M,L","in stock","padded"] |
| S002 | Shirts | V-Neck T-Shirt | ["type", "color", "fabric", "sizes"] | ["v-neck","white","blend","S,M,L,XL","in stock","discount - seasonal"] |
| S003 | Shirts | Tanktop | ["type", "color", "fabric", "sizes"] | ["tank","red","mesh","XS,S,M","discount - clearance","in stock"] |
| S004 | Shirts | Turtleneck | ["type", "color", "fabric", "sizes"] | ["turtle","black","cotton","M,L,XL","out of stock","padded"] |

**Transformation:**

When the above data is loaded into the Transformer page, you might need to clean up the two array columns.

Using the following transform, you can map the first element of the first array as a key for the first element of the second, which is its value. You might notice that the number of keys and the number of values are not consistent. For the extra elements in the second array, the default key of `ProdMiscProperties` is used:

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | ARRAYSTOMAP(ProdProperties, ProdValues, 'ProdMiscProperties') |
| Parameter: New column name | 'prodPropertyMap' |

You can now use the following steps to generate a new version of the keys:

| Transformation Name | Delete columns |
|---|---|
| Parameter: Columns | ProdKeys |
| Parameter: Action | Delete selected columns |

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | KEYS(prodPropertyMap) |
| Parameter: New column name | 'ProdKeys' |

**Results:**

| | | | | | |
|---|---|---|---|---|---|

| ProdId | ProdCategory | ProdName | ProdKeys | ProdProperties | prodPropertyMap |
|---|---|---|---|---|---|
| S001 | Shirts | Crew Neck T-Shirt | ["type", "color", "fabric", "sizes","ProdMiscProperties"] | ["crew","blue","cotton","S,M,L","in stock","padded"] | `{ "type": [ "crew" ], "color": [ "blue" ], "fabric": [ "cotton" ], "sizes": [ "S,M,L" ], "ProdMiscProperties": [ "in stock", "padded" ] }` |
| S002 | Shirts | V-Neck T-Shirt | ["type", "color", "fabric", "sizes","ProdMiscProperties"] | ["v-neck","white","blend","S,M,L,XL","in stock","discount - seasonal"] | `{ "type": [ "v-neck" ], "color": [ "white" ], "fabric": [ "blend" ], "sizes": [ "S,M,L,XL" ], "ProdMiscProperties": [ "in stock", "discount - seasonal" ] }` |
| S003 | Shirts | Tanktop | ["type", "color", "fabric", "sizes","ProdMiscProperties"] | ["tank","red","mesh","XS,S,M","discount - clearance","in stock"] | `{ "type": [ "tank" ], "color": [ "red" ], "fabric": [ "mesh" ], "sizes": [ "XS,S,M" ], "ProdMiscProperties": [ "discount - clearance", "in stock" ] }` |
| S004 | Shirts | Turtleneck | ["type", "color", "fabric", "sizes","ProdMiscProperties"] | ["turtle","black","cotton","M,L,XL","out of stock","padded"] | `{ "type": [ "turtle" ], "color": [ "black" ], "fabric": [ "cotton" ], "sizes": [ "M,L,XL" ], "ProdMiscProperties": [ "out of stock", "padded" ] }` |

# ARRAYUNIQUE Function

Generates an array of all unique elements among one or more arrays.

- Inputs are column names or array literals.
- If an element appears twice in one or more arrays, it is listed once in the output array.

**Wrangle vs. SQL:** This function is part of Wrangle , a proprietary data transformation language. Wrangle is not SQL. For more information, see *Wrangle Language*.

## Basic Usage

**Array literal reference example:**

```
arrayunique([["A","B"],["A","C"]])
```

**Output:** Returns a single array:

```
["A","B","C"]
```

**Single-column reference example:**

```
arrayunique([array1])
```

**Output:** Returns a single array of all unique elements in `array1` .

**Multi-column reference example:**

```
arrayunique([array1,array2])
```

**Output:** Returns a single array listing all unique elements in `array1` and `array2` .

## Syntax and Arguments

```
arrayunique(array_ref1,array_ref2)
```

| Argument | Required? | Data Type | Description |
|----------|-----------|-----------|-------------|
| array_ref1 | Y | string or array | Name of first column or first array literal to apply to the function |
| array_ref2 | N | string or array | Name of second column or second array literal to apply to the function |

For more information on syntax standards, see *Language Documentation Syntax Notes*.

**array_ref1, array_ref2**

Array literals or names of the array columns whose unique elements you want to derive.

**Usage Notes:**

| Required? | Data Type | Example Value |
|---|---|---|
| Yes (at least one) | Array literal or column reference | `myArray1,myArray2` |

## Examples

> **Tip:** For additional examples, see *Common Tasks*.

### Example - Simple unique example

This example illustrates how to use the nested functions.

**Functions**:

| Item | Description |
|---|---|
| ARRAYCONCAT Function | Combines the elements of one array with another, listing all elements of the first array before listing all elements of the second array. |
| ARRAYINTERSECT Function | Generates an array containing all elements that appear in multiple input arrays, referenced as column names or array literals. |
| ARRAYCROSS Function | Generates a nested array containing the cross-product of all elements in two or more arrays. |
| ARRAYUNIQUE Function | Generates an array of all unique elements among one or more arrays. |

**Source:**

Code formatting has been applied to improve legibility.

| Item | ArrayA | ArrayB |
|---|---|---|
| Item1 | `["A","B","C"]` | `["1","2","3"]` |
| Item2 | `["A","B"]` | `["A","B","C"]` |
| Item3 | `["D","E","F"]` | `["4","5","6"]` |

**Transformation:**

You can apply the following transforms in the following order. Note that the column names must be different from the transform name, which is a reserved word.

| Transformation Name | `New formula` |
|---|---|
| **Parameter: Formula type** | `Single row formula` |
| **Parameter: Formula** | `ARRAYCONCAT([Letters,Numerals])` |
| **Parameter: New column name** | `'concat2'` |

| Transformation Name | `New formula` |
|---|---|
| **Parameter: Formula type** | `Single row formula` |

| Parameter: Formula | ARRAYINTERSECT([Letters,Numerals]) |
|---|---|
| Parameter: New column name | 'intersection2' |

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | ARRAYCROSS([Letters,Numerals]) |
| Parameter: New column name | 'cross2' |

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | ARRAYUNIQUE([Letters,Numerals]) |
| Parameter: New column name | 'unique2' |

**Results:**

For display purposes, the results table has been broken down into three separate sets of columns.

Column set 1:

| Item | ArrayA | ArrayB | concat2 | intersection2 |
|---|---|---|---|---|
| Item1 | ["A","B","C"] | ["1","2","3"] | ["A","B","C","1","2","3"] | [ ] |
| Item2 | ["A","B"] | ["A","B","C"] | ["A","B","A","B","C"] | ["A","B"] |
| Item3 | ["D","E","F"] | ["4","5","6"] | ["D","E","F","4","5","6"] | [ ] |

Column set 2:

| Item | cross2 |
|---|---|
| Item1 | [["A","1"],["A","2"],["A","3"],["B","1"],["B","2"],["B","3"],["C","1"],["C","2"],["C","3"]] |
| Item2 | [["A","A"],["A","B"],["A","C"],["B","A"],["B","B"],["B","C"]] |
| Item3 | [["D","4"],["D","5"],["D","6"],["E","4"],["E","5"],["E","6"], ["F","4"],["F","5"],["F","6"]] |

Column set 3:

| Item | unique2 |
|---|---|
| Item1 | ["A","B","C","1","2","3"] |
| Item2 | ["A","B","C"] |
| Item3 | ["D","E","F","4","5","6"] |

# ARRAYZIP Function

**Contents:**

- *Basic Usage*
- *Syntax and Arguments*
    - *array_ref1, array_ref2*
- *Examples*
    - *Example - Simple ARRAYZIP example*
    - *Example - Unnest an array*

---

Combines multiple arrays into a single nested array, with element 1 of array 1 paired with element 2 of array 2 and so on. Arrays are expressed as column names or as array literals.

If the arrays are of different length, then null values are inserted for combinations where one array is missing a corresponding value.

**Wrangle vs. SQL:** This function is part of Wrangle , a proprietary data transformation language. Wrangle is not SQL. For more information, see *Wrangle Language*.

## Basic Usage

**Array literal reference example:**

```
arrayzip([["A","B","C"],["1","2","3"]])
```

**Output:** Returns a nested array combining elements from the two source arrays.

**Column reference example:**

```
arrayzip([array1,array2])
```

**Output:** Returns a single nested array pairing the elements of the array in the listed order of the arrays.

## Syntax and Arguments

```
arrayzip(array_ref1,array_ref2)
```

| Argument | Required? | Data Type | Description |
|----------|-----------|-----------|-------------|
| array_ref1 | Y | string or array | Name of first column or first array literal to apply to the function |
| array_ref2 | Y | string or array | Name of second column or second array literal to apply to the function |

For more information on syntax standards, see *Language Documentation Syntax Notes*.

**array_ref1, array_ref2**

Array literal or name of the array column whose elements you want to combine together.

**Usage Notes:**

| Required? | Data Type | Example Value |
|---|---|---|
| Yes | Array literal or column reference | `myArray1,myArray2` |

## Examples

> **Tip:** For additional examples, see *Common Tasks*.

### Example - Simple ARRAYZIP example

**Source:**

| Item | Letters | Numerals |
|---|---|---|
| Item1 | ["A","B","C"] | ["1","2","3"] |
| Item2 | ["D","E","F"] | ["4","5","6"] |
| Item3 | ["G","H","I"] | ["7","8","9"] |

**Transformation:**

| Transformation Name | `New formula` |
|---|---|
| **Parameter: Formula type** | `Single row formula` |
| **Parameter: Formula** | `arrayzip([Letters,Numerals])` |
| **Parameter: New column name** | `'LettersAndNumerals'` |

**Results:**

| Item | Letters | Numerals | LettersAndNumerals |
|---|---|---|---|
| Item1 | ["A","B","C"] | ["1","2","3"] | [["A","1"],["B",2],["C","3"]] |
| Item2 | ["D","E","F"] | ["4","5","6"] | [["F","4"],["G",5],["H","6"]] |
| Item3 | ["G","H","I"] | ["7","8","9"] | [["G","7"],["H",8],["I","9"]] |

### Example - Unnest an array

This example illustrates you to use the flatten and unnest transforms.

**Source:**

You have the following data on student test scores. Scores on individual scores are stored in the `Scores` array, and you need to be able to track each test on a uniquely identifiable row. This example has two goals:

1. One row for each student test
2. Unique identifier for each student-score combination

| LastName | FirstName | Scores |
|---|---|---|
| Adams | Allen | [81,87,83,79] |

| Burns | Bonnie | [98,94,92,85] |
| Cannon | Charles | [88,81,85,78] |

### Transformation:

When the data is imported from CSV format, you must add a `header` transform and remove the quotes from the `S cores` column:

| Transformation Name | Rename column with row(s) |
|---|---|
| **Parameter: Option** | Use row(s) as column names |
| **Parameter: Type** | Use a single row to name columns |
| **Parameter: Row number** | 1 |

| Transformation Name | Replace text or pattern |
|---|---|
| **Parameter: Column** | colScores |
| **Parameter: Find** | `'\"'` |
| **Parameter: Replace with** | `''` |
| **Parameter: Match all occurrences** | true |

**Validate test date:** To begin, you might want to check to see if you have the proper number of test scores for each student. You can use the following transform to calculate the difference between the expected number of elements in the `Scores` array (4) and the actual number:

| Transformation Name | New formula |
|---|---|
| **Parameter: Formula type** | Single row formula |
| **Parameter: Formula** | (4 - arraylen(Scores)) |
| **Parameter: New column name** | 'numMissingTests' |

When the transform is previewed, you can see in the sample dataset that all tests are included. You might or might not want to include this column in the final dataset, as you might identify missing tests when the recipe is run at scale.

**Unique row identifier:** The `Scores` array must be broken out into individual rows for each test. However, there is no unique identifier for the row to track individual tests. In theory, you could use the combination of `LastName-FirstName-Scores` values to do so, but if a student recorded the same score twice, your dataset has duplicate rows. In the following transform, you create a parallel array called `Tests`, which contains an index array for the number of values in the `Scores` column. Index values start at `0`:

| Transformation Name | New formula |
|---|---|
| **Parameter: Formula type** | Single row formula |
| **Parameter: Formula** | range(0,arraylen(Scores)) |
| **Parameter: New column name** | 'Tests' |

Also, we will want to create an identifier for the source row using the `sourcerownumber` function:

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | sourcerownumber() |
| Parameter: New column name | 'orderIndex' |

**One row for each student test:** Your data should look like the following:

| LastName | FirstName | Scores | Tests | orderIndex |
|---|---|---|---|---|
| Adams | Allen | [81,87,83,79] | [0,1,2,3] | 2 |
| Burns | Bonnie | [98,94,92,85] | [0,1,2,3] | 3 |
| Cannon | Charles | [88,81,85,78] | [0,1,2,3] | 4 |

Now, you want to bring together the `Tests` and `Scores` arrays into a single nested array using the `arrayzip` function:

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | arrayzip([Tests,Scores]) |

Your dataset has been changed:

| LastName | FirstName | Scores | Tests | orderIndex | column1 |
|---|---|---|---|---|---|
| Adams | Allen | [81,87,83,79] | [0,1,2,3] | 2 | [[0,81],[1,87],[2,83],[3,79]] |
| Adams | Bonnie | [98,94,92,85] | [0,1,2,3] | 3 | [[0,98],[1,94],[2,92],[3,85]] |
| Cannon | Charles | [88,81,85,78] | [0,1,2,3] | 4 | [[0,88],[1,81],[2,85],[3,78]] |

Use the following to unpack the nested array:

| Transformation Name | Expand arrays to rows |
|---|---|
| Parameter: Column | column1 |

Each test-score combination is now broken out into a separate row. The nested Test-Score combinations must be broken out into separate columns using the following:

| Transformation Name | Unnest Objects into columns |
|---|---|
| Parameter: Column | column1 |
| Parameter: Paths to elements | '[0]','[1]' |

After you delete `column1`, which is no longer needed you should rename the two generated columns:

| Transformation Name | Rename columns |
|---|---|
| | |

| | |
|---|---|
| **Parameter: Option** | `Manual rename` |
| **Parameter: Column** | `column_0` |
| **Parameter: New column name** | `'TestNum'` |

| | |
|---|---|
| **Transformation Name** | `Rename columns` |
| **Parameter: Option** | `Manual rename` |
| **Parameter: Column** | `column_1` |
| **Parameter: New column name** | `'TestScore'` |

**Unique row identifier:** You can do one more step to create unique test identifiers, which identify the specific test for each student. The following uses the original row identifier `OrderIndex` as an identifier for the student and the `TestNumber` value to create the `TestId` column value:

| | |
|---|---|
| **Transformation Name** | `New formula` |
| **Parameter: Formula type** | `Single row formula` |
| **Parameter: Formula** | `(orderIndex * 10) + TestNum` |
| **Parameter: New column name** | `'TestId'` |

The above are integer values. To make your identifiers look prettier, you might add the following:

| | |
|---|---|
| **Transformation Name** | `Merge columns` |
| **Parameter: Columns** | `'TestId00','TestId'` |

**Extending:** You might want to generate some summary statistical information on this dataset. For example, you might be interested in calculating each student's average test score. This step requires figuring out how to properly group the test values. In this case, you cannot group by the `LastName` value, and when executed at scale, there might be collisions between first names when this recipe is run at scale. So, you might need to create a kind of primary key using the following:

| | |
|---|---|
| **Transformation Name** | `Merge columns` |
| **Parameter: Columns** | `'LastName','FirstName'` |
| **Parameter: Separator** | `'-'` |
| **Parameter: New column name** | `'studentId'` |

You can now use this as a grouping parameter for your calculation:

| | |
|---|---|
| **Transformation Name** | `New formula` |
| **Parameter: Formula type** | `Single row formula` |
| **Parameter: Formula** | `average(TestScore)` |
| **Parameter: Group rows by** | `studentId` |
| | |

| Parameter: New column name | `'avg_TestScore'` |
|---|---|

**Results:**

After you delete unnecessary columns and move your columns around, the dataset should look like the following:

| TestId | LastName | FirstName | TestNum | TestScore | studentId | avg_TestScore |
|---|---|---|---|---|---|---|
| TestId0021 | Adams | Allen | 0 | 81 | Adams-Allen | 82.5 |
| TestId0022 | Adams | Allen | 1 | 87 | Adams-Allen | 82.5 |
| TestId0023 | Adams | Allen | 2 | 83 | Adams-Allen | 82.5 |
| TestId0024 | Adams | Allen | 3 | 79 | Adams-Allen | 82.5 |
| TestId0031 | Adams | Bonnie | 0 | 98 | Adams-Bonnie | 92.25 |
| TestId0032 | Adams | Bonnie | 1 | 94 | Adams-Bonnie | 92.25 |
| TestId0033 | Adams | Bonnie | 2 | 92 | Adams-Bonnie | 92.25 |
| TestId0034 | Adams | Bonnie | 3 | 85 | Adams-Bonnie | 92.25 |
| TestId0041 | Cannon | Chris | 0 | 88 | Cannon-Chris | 83 |
| TestId0042 | Cannon | Chris | 1 | 81 | Cannon-Chris | 83 |
| TestId0043 | Cannon | Chris | 2 | 85 | Cannon-Chris | 83 |
| TestId0044 | Cannon | Chris | 3 | 78 | Cannon-Chris | 83 |

# FILTEROBJECT Function

**Contents:**

- *Basic Usage*
- *Syntax and Arguments*
  - *obj_col*
  - *keys*
- *Examples*
  - *Example - Parsing query parameters from URLs*

---

Filters the keys and values from an Object data type column based on a specified key value.

- A single field value of an Object data type must have unique keys. Values may, however, be repeated.
- The order of key-value pairs is not guaranteed.

**Wrangle vs. SQL:** This function is part of Wrangle , a proprietary data transformation language. Wrangle is not SQL. For more information, see *Wrangle Language*.

## Basic Usage

**Object literal reference example:**

```
filterobject('{"q":"hello","r","there":"q","world"}', 'q')
```

**Output:** Returns an Object of key-value pairs for the q key:

```
{"q":["hello", "world"]}
```

**Column reference example:**

```
filterobject(myObjects, '[k1,k2]')
```

**Output:** Returns an Object of key-value pairs for all instances of the k1 and k2 keys.

## Syntax and Arguments

```
filterobject(obj,'keys')
```

| Argument | Required? | Data Type | Description |
|----------|-----------|-----------|-------------|
| obj_col | Y | String or Object | Name of column, function returning an Object, or Object literal to be filtered |
| keys | Y | Array | Array representing the keys to filter. Each element can be a String, function returning a String, or a reference to a column of String values. |

For more information on syntax standards, see *Language Documentation Syntax Notes*.

**obj_col**

Object literal, name of the Object column, or function returning an Object whose keys you want to extract into an array.

**Usage Notes:**

| Required? | Data Type | Example Value |
|-----------|-----------|---------------|
| Yes | Object literal, function, or column reference | `myObj` |

**keys**

This parameter contains an Array of Strings, each of which represents a key whose values are to be returned with the key as the output of the function.

- For a single key, this value can be a regular String value.
- For multiple keys, this value is an Array of String values.

**Usage Notes:**

| Required? | Data Type | Example Value |
|-----------|-----------|---------------|
| Yes | String or Array | `['key1','key2','key3']` |

## Examples

> **Tip:** For additional examples, see *Common Tasks*.

**Example - Parsing query parameters from URLs**

This examples illustrates how you can extract component parts of a URL using specialized functions for the URL data type.

**Functions**:

| Item | Description |
|------|-------------|
| DOMAIN Function | Finds the value for the domain from a valid URL. Input values must be of URL or String type. |
| SUBDOMAIN Function | Finds the value a subdomain value from a valid URL. Input values must be of URL or String type. |
| HOST Function | Finds the host value from a valid URL. Input values must be of URL or String type and can be literals or column references. |
| SUFFIX Function | Finds the suffix value after the domain from a valid URL. Input values must be of URL or String type. |
| URLPARAMS Function | Extracts the query parameters of a URL into an Object. The Object keys are the parameter's names, and its values are the parameter's values. Input values must be of URL or String type. |
| FILTEROBJECT Function | Filters the keys and values from an Object data type column based on a specified key value. |

**Source:**

Your dataset includes the following values for URLs:

| URL |
| --- |
| www.example.com |
| example.com/support |
| http://www.example.com/products/ |
| http://1.2.3.4 |
| https://www.example.com/free-download |
| https://www.example.com/about-us/careers |
| www.app.example.com |
| www.some.app.example.com |
| some.app.example.com |
| some.example.com |
| example.com |
| http://www.example.com?q1=broken%20record |
| http://www.example.com?query=khakis&app=pants |
| http://www.example.com?q1=broken%20record&q2=broken%20tape&q3=broken%20wrist |

**Transformation:**

When the above data is imported into the application, the column is recognized as a URL. All values are registered as valid, even the numeric address.

To extract the domain and subdomain values:

| | |
| --- | --- |
| **Transformation Name** | New formula |
| **Parameter: Formula type** | Single row formula |
| **Parameter: Formula** | DOMAIN(URL) |
| **Parameter: New column name** | 'domain_URL' |

| | |
| --- | --- |
| **Transformation Name** | New formula |
| **Parameter: Formula type** | Single row formula |
| **Parameter: Formula** | SUBDOMAIN(URL) |
| **Parameter: New column name** | 'subdomain_URL' |

| | |
| --- | --- |
| **Transformation Name** | New formula |
| **Parameter: Formula type** | Single row formula |
| **Parameter: Formula** | HOST(URL) |
| **Parameter: New column name** | 'host_URL' |

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | SUFFIX(URL) |
| Parameter: New column name | 'suffix_URL' |

You can use the Pattern in the following transformation to extract protocol identifiers, if present, into a new column:

| Transformation Name | Extract text or pattern |
|---|---|
| Parameter: Column to extract from | URL |
| Parameter: Option | Custom text or pattern |
| Parameter: Text to extract | `` `{start}%*://` `` |

To clean this up, you might want to rename the column to `protocol_URL`.

To extract the path values, you can use the following regular expression:

> **NOTE:** Regular expressions are considered a developer-level method for pattern matching. Please use them with caution. See *Text Matching*.

| Transformation Name | Extract text or pattern |
|---|---|
| Parameter: Column to extract from | URL |
| Parameter: Option | Custom text or pattern |
| Parameter: Text to extract | /[^*:\/\/]\/.*$/ |

The above transformation grabs a little too much of the URL. If you rename the column to `path_URL`, you can use the following regular expression to clean it up:

| Transformation Name | Extract text or pattern |
|---|---|
| Parameter: Column to extract from | URL |
| Parameter: Option | Custom text or pattern |
| Parameter: Text to extract | /[!^\/].*$/ |

Delete the `path_URL` column and rename the `path_URL1` column to the deleted one. Then:

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | URLPARAMS(URL) |
| Parameter: New column | 'urlParams' |

If you wanted to just see the values for the `q1` parameter, you could add the following:

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | FILTEROBJECT(urlParams,'q1') |
| Parameter: New column name | 'urlParam_q1' |

**Results:**

For display purposes, the results table has been broken down into separate sets of columns.

Column set 1:

| URL | host_URL | path_URL |
|---|---|---|
| www.example.com | www.example.com | |
| example.com/support | example.com | /support |
| http://www.example.com/products/ | www.example.com | /products/ |
| http://1.2.3.4 | 1.2.3.4 | |
| https://www.example.com/free-download | www.example.com | /free-download |
| https://www.example.com/about-us/careers | www.example.com | /about-us /careers |
| www.app.example.com | www.app.example.com | |
| www.some.app.example.com | www.some.app.example. com | |
| some.app.example.com | some.app.example.com | |
| some.example.com | some.example.com | |
| example.com | example.com | |
| http://www.example.com?q1=broken%20record | www.example.com | |
| http://www.example.com?query=khakis&app=pants | www.example.com | |
| http://www.example.com?q1=broken%20record&q2=broken%20tape&q3=broken%20wrist | www.example.com | |

Column set 2:

| URL | protocol_URL | subdomain_URL | domain_URL | suffix_URL |
|---|---|---|---|---|
| www.example.com | | www | example | com |
| example.com/support | | | example | com |
| http://www.example.com/products/ | http:// | www | example | com |
| http://1.2.3.4 | http:// | | | |
| https://www.example.com/free-download | https:// | www | example | com |
| https://www.example.com/about-us/careers | https:// | www | example | com |
| www.app.example.com | | www.app | example | com |

| URL | | | | |
|---|---|---|---|---|
| www.some.app.example.com | | www.some.app | example | com |
| some.app.example.com | | some.app | example | com |
| some.example.com | | some | example | com |
| example.com | | | example | com |
| http://www.example.com?q1=broken%20record | http:// | www | example | com |
| http://www.example.com?query=khakis&app=pants | http:// | www | example | com |
| http://www.example.com?q1=broken%20record&q2=broken%20tape&q3=broken%20wrist | http:// | www | example | com |

Column set 3:

| URL | urlParams | urlParam_q1 |
|---|---|---|
| www.example.com | | |
| example.com/support | | |
| http://www.example.com/products/ | | |
| http://1.2.3.4 | | |
| https://www.example.com/free-download | | |
| https://www.example.com/about-us/careers | | |
| www.app.example.com | | |
| www.some.app.example.com | | |
| some.app.example.com | | |
| some.example.com | | |
| example.com | | |
| http://www.example.com?q1=broken%20record | {"q1":"broken record"} | {"q1":"broken record"} |
| http://www.example.com?query=khakis&app=pants | {"query":"khakis","app":"pants"} | |
| http://www.example.com?q1=broken%20record&q2=broken%20tape&q3=broken%20wrist | {"q1":"broken record", "q2":"broken tape", "q3":"broken wrist"} | {"q1":"broken record"} |

# KEYS Function

**Contents:**

---

Extracts the key values from an Object data type column and stores them in an array of String values.

**Wrangle vs. SQL:** This function is part of Wrangle , a proprietary data transformation language. Wrangle is not SQL. For more information, see *Wrangle Language*.

## Basic Usage

**Column reference example:**

```
keys(&apos;object1&apos;)
```

**Output:** Returns an array of all of the keys found in the key-value Objects found in the `object1` column.

## Syntax and Arguments

```
keys(obj_col)
```

| Argument | Required? | Data Type | Description |
|----------|-----------|-----------|-------------|
| obj_col | Y | String or Object | Name of column or Object literal whose keys are to be extracted into an array |

For more information on syntax standards, see *Language Documentation Syntax Notes*.

**obj_col**

Object literal or name of the Object column whose keys you want to extract into an array.

**Usage Notes:**

| Required? | Data Type | Example Value |
|-----------|-----------|---------------|
| Yes | Object literal or column reference | `myObj` |

## Examples

> **Tip:** For additional examples, see *Common Tasks*.

## Example - Basic keys example

### Source:

Following dataset contains configuration blocks for individual features. These example blocks are of Object type.

Code formatting has been applied to the Object data to improve legibility.

| FeatureName | Configuration |
|---|---|
| Whiz Widget | ```<br>{<br> "enabled": "true",<br> "maxRows": "1000",<br> "maxCols": "100"<br>}<br>``` |
| Magic Button | ```<br>{<br> "enabled": "false",<br> "startDirectory": "/home",<br> "maxDepth": "15"<br>}<br>``` |
| Happy Path Finder | ```<br>{<br> "enabled": "true"<br>}<br>``` |

### Transformation:

The following transformation extracts the key values from the Object data in the `Configuration` column.

| Transformation Name | `New formula` |
|---|---|
| Parameter: Formula type | `Single row formula` |
| Parameter: Formula | `keys(Configuration)` |
| Parameter: New column name | `'keys_Configuration'` |

### Results:

The `keys_Configuration` column contains the arrays of the key values.

| FeatureName | Configuration | keys_Configuration |
|---|---|---|
| Whiz Widget | ```<br>{<br> "enabled": "true",<br> "maxRows": "1000",<br> "maxCols": "100"<br>}<br>``` | `["enabled","maxRows","maxCols"]` |
| Magic Button | ```<br>{<br> "enabled": "false",<br>``` | `["enabled","startDirectory","maxDepth"]` |

| | | |
|---|---|---|
| | ```
    "startDirectory": "/home",
    "maxDepth": "15"
}
``` | |
| Happy Path Finder | ```
{
    "enabled": "true"
}
``` | `["enabled"]` |

### Example - Create an Object of product properties

This example illustrates how to use the ARRAYSTOMAP and KEYS functions to convert values in Array or Object data type of key-value pairs.

#### Functions:

| Item | Description |
|---|---|
| ARRAYSTOMAP Function | Combines one array containing keys and another array containing values into an Object of key-value pairs. |
| KEYS Function | Extracts the key values from an Object data type column and stores them in an array of String values. |

#### Source:

Your dataset contains master product data with product properties stored in two arrays of keys and values.

| ProdId | ProdCategory | ProdName | ProdKeys | ProdProperties |
|---|---|---|---|---|
| S001 | Shirts | Crew Neck T-Shirt | ["type", "color", "fabric", "sizes"] | ["crew","blue","cotton","S,M,L","in stock","padded"] |
| S002 | Shirts | V-Neck T-Shirt | ["type", "color", "fabric", "sizes"] | ["v-neck","white","blend","S,M,L,XL","in stock","discount - seasonal"] |
| S003 | Shirts | Tanktop | ["type", "color", "fabric", "sizes"] | ["tank","red","mesh","XS,S,M","discount - clearance","in stock"] |
| S004 | Shirts | Turtleneck | ["type", "color", "fabric", "sizes"] | ["turtle","black","cotton","M,L,XL","out of stock","padded"] |

#### Transformation:

When the above data is loaded into the Transformer page, you might need to clean up the two array columns.

Using the following transform, you can map the first element of the first array as a key for the first element of the second, which is its value. You might notice that the number of keys and the number of values are not consistent. For the extra elements in the second array, the default key of `ProdMiscProperties` is used:

| Transformation Name | `New formula` |
|---|---|
| **Parameter: Formula type** | `Single row formula` |
| **Parameter: Formula** | `ARRAYSTOMAP(ProdProperties, ProdValues, 'ProdMiscProperties')` |
| **Parameter: New column name** | `'prodPropertyMap'` |

You can now use the following steps to generate a new version of the keys:

| Transformation Name | Delete columns |
| --- | --- |
| Parameter: Columns | ProdKeys |
| Parameter: Action | Delete selected columns |

| Transformation Name | New formula |
| --- | --- |
| Parameter: Formula type | Single row formula |
| Parameter: Formula | KEYS(prodPropertyMap) |
| Parameter: New column name | 'ProdKeys' |

**Results:**

| ProdId | ProdCategory | ProdName | ProdKeys | ProdProperties | prodPropertyMap |
| --- | --- | --- | --- | --- | --- |
| S001 | Shirts | Crew Neck T-Shirt | ["type", "color", "fabric", "sizes","ProdMiscProperties"] | ["crew","blue","cotton","S,M,L","in stock","padded"] | {<br>  "type": [ "crew" ],<br>  "color": [ "blue" ],<br>  "fabric": [ "cotton" ],<br>  "sizes": [ "S,M,L" ],<br>  "ProdMiscProperties": [ "in stock", "padded" ] } |
| S002 | Shirts | V-Neck T-Shirt | ["type", "color", "fabric", "sizes","ProdMiscProperties"] | ["v-neck","white","blend","S,M,L,XL"," in stock","discount - seasonal"] | {<br>  "type": [ "v-neck" ],<br>  "color": [ "white" ],<br>  "fabric": [ "blend" ],<br>  "sizes": [ "S,M,L,XL" ],<br>  "ProdMiscProperties": [ "in stock", "discount - seasonal" ] } |
| S003 | Shirts | Tanktop | ["type", "color", "fabric", "sizes","ProdMiscProperties"] | ["tank","red","mesh","XS,S,M"," discount - clearance","in stock"] | {<br>  "type": [ "tank" ],<br>  "color": [ "red" ],<br>  "fabric": [ "mesh" ],<br>  "sizes": [ "XS,S,M" ],<br>  "ProdMiscProperties": [ "discount - clearance", "in stock" ] } |
| S004 | Shirts | Turtleneck | ["type", "color", "fabric", "sizes","ProdMiscProperties"] | ["turtle","black","cotton","M,L,XL","out of stock","padded"] | {<br>  "type": [ "turtle" ],<br>  "color": [ "black" ], |

| | | | | | "fabric": [ "cotton" ],<br>  "sizes": [ "M,L,XL" ],<br>  "ProdMiscProperties": [ "out of stock", "padded" ] } |
|---|---|---|---|---|---|

# LISTAVERAGE Function

Computes the average of all numeric values found in input array. Input can be an array literal, a column of arrays, or a function returning an array. Input values must be of Integer or Decimal type.

When this function is invoked, all of the values in the input array are passed to the corresponding columnar function. Some restrictions may apply. See *AVERAGE Function*.

**Wrangle vs. SQL:** This function is part of Wrangle , a proprietary data transformation language. Wrangle is not SQL. For more information, see *Wrangle Language*.

## Basic Usage

**Literal example:**

```
listaverage([0,0,2,4,6,8,10,12,14,16,18,20])
```

**Output:** Returns the average of all values in the literal array: `19.1667`.

**Column example:**

```
listaverage(myArray)
```

**Output:** Returns the average of all values in the arrays of the `myArray` column.

## Syntax and Arguments

```
listaverage(array_ref)
```

| Argument | Required? | Data Type | Description |
|---|---|---|---|
| array_ref | Y | Array | Array literal, reference to column containing arrays, or function returning an array |

For more information on syntax standards, see *Language Documentation Syntax Notes*.

**array_ref**

Reference to an array can be an array literal, function returning an array, or a single column containing arrays.

- If the input is not a valid numeric array, null values are returned.
- Non-numerical values within an input array are not factored in the computation.
- Multiple columns and wildcards are not supported.

**Usage Notes:**

| Required? | Data Type | Example Value |
|---|---|---|
| Yes | Array | `myArray` |

## Examples

> **Tip:** For additional examples, see *Common Tasks*.

### Example - Math functions for lists (arrays)

This example describes how to generate random array (list) data and then to apply statistical functions specifically created for arrays.

**Functions:**

| Item | Description |
|------|-------------|
| LISTSUM Function | Computes the sum of all numeric values found in input array. Input can be an array literal, a column of arrays, or a function returning an array. Input values must be of Integer or Decimal type. |
| LISTMIN Function | Computes the minimum of all numeric values found in input array. Input can be an array literal, a column of arrays, or a function returning an array. Input values must be of Integer or Decimal type. |
| LISTMAX Function | Computes the maximum of all numeric values found in input array. Input can be an array literal, a column of arrays, or a function returning an array. Input values must be of Integer or Decimal type. |
| LISTAVERAGE Function | Computes the average of all numeric values found in input array. Input can be an array literal, a column of arrays, or a function returning an array. Input values must be of Integer or Decimal type. |
| LISTVAR Function | Computes the variance of all numeric values found in input array. Input can be an array literal, a column of arrays, or a function returning an array. Input values must be of Integer or Decimal type. |
| LISTSTDEV Function | Computes the standard deviation of all numeric values found in input array. Input can be an array literal, a column of arrays, or a function returning an array. Input values must be of Integer or Decimal type. |
| LISTMODE Function | Computes the most common value of all numeric values found in input array. Input can be an array literal, a column of arrays, or a function returning an array. Input values must be of Integer or Decimal type. |

Also:

| Item | Description |
|------|-------------|
| RANGE Function | Computes an array of integers, from a beginning integer to an end (stop) integer, stepping by a third parameter. |
| RAND Function | The RAND function generates a random real number between 0 and 1. The function accepts an optional integer parameter, which causes the same set of random numbers to be generated with each job execution. |
| ROUND Function | Rounds input value to the nearest integer. Input can be an Integer, a Decimal, a column reference, or an expression. Optional second argument can be used to specify the number of digits to which to round. |

**Source:**

For this example, you can generate some randomized data using the following steps. First, you need to seed an array with a range of values using the RANGE function:

| | |
|------|------|
| **Transformation Name** | New formula |
| **Parameter: Formula type** | Single row formula |
| **Parameter: Formula** | RANGE(5, 50, 5) |
| **Parameter: New column name** | 'myArray1' |

Then, unpack this array, so you can add a random factor:

| | |
|------|------|
| **Transformation Name** | Unnest Objects into columns |
| **Parameter: Column** | myArray1 |

| Parameter: Paths to elements | '[0]', '[1]', '[2]', '[3]', '[4]', '[5]', '[6]', '[7]', '[8]', '[9]' |
|---|---|
| Parameter: Remove elements from original | true |
| Parameter: Include original column name | true |

Add the randomizing factor. Here, you are adding randomization around individual values:  x-1 < x < x+4.

| Transformation Name | Edit column with formula |
|---|---|
| Parameter: Columns | myArray1_0~myArray1_8 |
| Parameter: Formula | IF(RAND() > 0.5, $col + (5 * RAND()), $col - RAND()) |

To make the numbers easier to manipulate, you can round them to two decimal places:

| Transformation Name | Edit column with formula |
|---|---|
| Parameter: Columns | myArray1_0~myArray1_8 |
| Parameter: Formula | ROUND($col, 2) |

Renest these columns into an array:

| Transformation Name | Nest columns into Objects |
|---|---|
| Parameter: Columns | myArray1_0, myArray1_1, myArray1_2, myArray1_3, myArray1_4, myArray1_5, myArray1_6, myArray1_7, myArray1_8 |
| Parameter: Nest columns to | Array |
| Parameter: New column name | 'myArray2' |

Delete the unused columns:

| Transformation Name | Delete columns |
|---|---|
| Parameter: Columns | myArray1_0~myArray1_8,myArray1 |
| Parameter: Action | Delete selected columns |

Your data should look similar to the following:

| myArray2 |
|---|
| ["8.29","9.63","14.63","19.63","24.63","29.63","34.63","39.63","44.63"] |
| ["8.32","14.01","19.01","24.01","29.01","34.01","39.01","44.01","49.01"] |
| ["4.55","9.58","14.58","19.58","24.58","29.58","34.58","39.58","44.58"] |
| ["9.22","14.84","19.84","24.84","29.84","34.84","39.84","44.84","49.84"] |
| ["8.75","13.36","18.36","23.36","28.36","33.36","38.36","43.36","48.36"] |
| ["8.47","14.76","19.76","24.76","29.76","34.76","39.76","44.76","49.76"] |

| | |
|---|---|
| ["4.93","9.99","14.99","19.99","24.99","29.99","34.99","39.99","44.99"] |
| ["4.65","14.98","19.98","24.98","29.98","34.98","39.98","44.98","49.98"] |
| ["7.80","14.62","19.62","24.62","29.62","34.62","39.62","44.62","49.62"] |
| ["9.32","9.96","14.96","19.96","24.96","29.96","34.96","39.96","44.96"] |

**Transformation:**

These steps demonstrate the individual math functions that you can apply to your list data without unnesting it:

> **NOTE:** The NUMFORMAT function has been wrapped around each list function to account for any floating-point errors or additional digits in the results.

Sum of all values in the array (list):

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | NUMFORMAT(LISTSUM(myArray2), '#.##') |
| Parameter: New column name | 'arraySum' |

Minimum of all values in the array (list):

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | NUMFORMAT(LISTMIN(myArray2), '#.##') |
| Parameter: New column name | 'arrayMin' |

Maximum of all values in the array (list):

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | NUMFORMAT(LISTMAX(myArray2), '#.##') |
| Parameter: New column name | 'arrayMax' |

Average of all values in the array (list):

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | NUMFORMAT(LISTAVERAGE(myArray2), '#.##') |
| Parameter: New column name | 'arrayAvg' |

Variance of all values in the array (list):

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | NUMFORMAT(LISTVAR(myArray2), '#.##') |
| Parameter: New column name | 'arrayVar' |

Standard deviation of all values in the array (list):

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | NUMFORMAT(LISTSTDEV(myArray2), '#.##') |
| Parameter: New column name | 'arrayStDv' |

Mode (most common value) of all values in the array (list):

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | NUMFORMAT(LISTMODE(myArray2), '#.##') |
| Parameter: New column name | 'arrayMode' |

**Results:**

Results for the first four math functions:

| myArray2 | arrayAvg | arrayMax | arrayMin | arraySum |
|---|---|---|---|---|
| ["8.29","9.63","14.63","19.63","24.63","29.63","34.63","39.63","44.63"] | 25.04 | 44.63 | 8.29 | 225.33 |
| ["8.32","14.01","19.01","24.01","29.01","34.01","39.01","44.01","49.01"] | 28.93 | 49.01 | 8.32 | 260.4 |
| ["4.55","9.58","14.58","19.58","24.58","29.58","34.58","39.58","44.58"] | 24.58 | 44.58 | 4.55 | 221.19 |
| ["9.22","14.84","19.84","24.84","29.84","34.84","39.84","44.84","49.84"] | 29.77 | 49.84 | 9.22 | 267.94 |
| ["8.75","13.36","18.36","23.36","28.36","33.36","38.36","43.36","48.36"] | 28.4 | 48.36 | 8.75 | 255.63 |
| ["8.47","14.76","19.76","24.76","29.76","34.76","39.76","44.76","49.76"] | 29.62 | 49.76 | 8.47 | 266.55 |
| ["4.93","9.99","14.99","19.99","24.99","29.99","34.99","39.99","44.99"] | 24.98 | 44.99 | 4.93 | 224.85 |
| ["4.65","14.98","19.98","24.98","29.98","34.98","39.98","44.98","49.98"] | 29.39 | 49.98 | 4.65 | 264.49 |
| ["7.80","14.62","19.62","24.62","29.62","34.62","39.62","44.62","49.62"] | 29.42 | 49.62 | 7.8 | 264.76 |
| ["9.32","9.96","14.96","19.96","24.96","29.96","34.96","39.96","44.96"] | 25.44 | 44.96 | 9.32 | 229 |

Results for the statistical functions:

| myArray2 | arrayMode | arrayStDv | arrayVar |
|---|---|---|---|

| | | | |
|---|---|---|---|
| ["8.29","9.63","14.63","19.63","24.63","29.63","34.63","39.63","44.63"] | | 12.32 | 151.72 |
| ["8.32","14.01","19.01","24.01","29.01","34.01","39.01","44.01","49.01"] | | 13.03 | 169.78 |
| ["4.55","9.58","14.58","19.58","24.58","29.58","34.58","39.58","44.58"] | | 12.92 | 166.8 |
| ["9.22","14.84","19.84","24.84","29.84","34.84","39.84","44.84","49.84"] | | 13.02 | 169.46 |
| ["8.75","13.36","18.36","23.36","28.36","33.36","38.36","43.36","48.36"] | | 12.84 | 164.95 |
| ["8.47","14.76","19.76","24.76","29.76","34.76","39.76","44.76","49.76"] | | 13.14 | 172.56 |
| ["4.93","9.99","14.99","19.99","24.99","29.99","34.99","39.99","44.99"] | | 12.92 | 166.93 |
| ["4.65","14.98","19.98","24.98","29.98","34.98","39.98","44.98","49.98"] | | 13.9 | 193.16 |
| ["7.80","14.62","19.62","24.62","29.62","34.62","39.62","44.62","49.62"] | | 13.23 | 175.08 |
| ["9.32","9.96","14.96","19.96","24.96","29.96","34.96","39.96","44.96"] | | 12.21 | 149.17 |

Since all values are unique within an individual array, there is no most common value in any of them, which yields empty values for the `arrayMode` column.

# LISTMAX Function

Computes the maximum of all numeric values found in input array. Input can be an array literal, a column of arrays, or a function returning an array. Input values must be of Integer or Decimal type.

When this function is invoked, all of the values in the input array are passed to the corresponding columnar function. Some restrictions may apply. See *MAX Function*.

**Wrangle vs. SQL:** This function is part of Wrangle , a proprietary data transformation language. Wrangle is not SQL. For more information, see *Wrangle Language*.

## Basic Usage

**Literal example:**

```
listmax([0,0,2,4,6,8,10,12,14,16,18,20])
```

**Output:** Returns the maximum of all values in the literal array: `20`.

**Column example:**

```
listmax(myArray)
```

**Output:** Generates an output column containing the maximum of all values in the arrays of the `myArray` column.

## Syntax and Arguments

```
listmax(array_ref)
```

| Argument | Required? | Data Type | Description |
|----------|-----------|-----------|-------------|
| array_ref | Y | Array | Array literal, reference to column containing arrays, or function returning an array |

For more information on syntax standards, see *Language Documentation Syntax Notes*.

**array_ref**

Reference to an array can be an array literal, function returning an array, or a single column containing arrays.

- If the input is not a valid numeric array, null values are returned.
- Non-numerical values within an input array are not factored in the computation.
- Multiple columns and wildcards are not supported.

**Usage Notes:**

| Required? | Data Type | Example Value |
|-----------|-----------|---------------|
| Yes | Array | `myArray` |

## Examples

> **Tip:** For additional examples, see *Common Tasks*.

## Example - Math functions for lists (arrays)

This example describes how to generate random array (list) data and then to apply statistical functions specifically created for arrays.

**Functions:**

| Item | Description |
|------|-------------|
| LISTSUM Function | Computes the sum of all numeric values found in input array. Input can be an array literal, a column of arrays, or a function returning an array. Input values must be of Integer or Decimal type. |
| LISTMIN Function | Computes the minimum of all numeric values found in input array. Input can be an array literal, a column of arrays, or a function returning an array. Input values must be of Integer or Decimal type. |
| LISTMAX Function | Computes the maximum of all numeric values found in input array. Input can be an array literal, a column of arrays, or a function returning an array. Input values must be of Integer or Decimal type. |
| LISTAVERAGE Function | Computes the average of all numeric values found in input array. Input can be an array literal, a column of arrays, or a function returning an array. Input values must be of Integer or Decimal type. |
| LISTVAR Function | Computes the variance of all numeric values found in input array. Input can be an array literal, a column of arrays, or a function returning an array. Input values must be of Integer or Decimal type. |
| LISTSTDEV Function | Computes the standard deviation of all numeric values found in input array. Input can be an array literal, a column of arrays, or a function returning an array. Input values must be of Integer or Decimal type. |
| LISTMODE Function | Computes the most common value of all numeric values found in input array. Input can be an array literal, a column of arrays, or a function returning an array. Input values must be of Integer or Decimal type. |

Also:

| Item | Description |
|------|-------------|
| RANGE Function | Computes an array of integers, from a beginning integer to an end (stop) integer, stepping by a third parameter. |
| RAND Function | The RAND function generates a random real number between 0 and 1. The function accepts an optional integer parameter, which causes the same set of random numbers to be generated with each job execution. |
| ROUND Function | Rounds input value to the nearest integer. Input can be an Integer, a Decimal, a column reference, or an expression. Optional second argument can be used to specify the number of digits to which to round. |

**Source:**

For this example, you can generate some randomized data using the following steps. First, you need to seed an array with a range of values using the RANGE function:

| | |
|------|-------------|
| **Transformation Name** | `New formula` |
| **Parameter: Formula type** | `Single row formula` |
| **Parameter: Formula** | `RANGE(5, 50, 5)` |
| **Parameter: New column name** | `'myArray1'` |

Then, unpack this array, so you can add a random factor:

| | |
|------|-------------|
| **Transformation Name** | `Unnest Objects into columns` |
| **Parameter: Column** | `myArray1` |
| **Parameter: Paths to elements** | `'[0]', '[1]', '[2]', '[3]', '[4]', '[5]', '[6]',` |

| | '[7]', '[8]', '[9]' |
|---|---|
| **Parameter: Remove elements from original** | true |
| **Parameter: Include original column name** | true |

Add the randomizing factor. Here, you are adding randomization around individual values: x-1 < x < x+4.

| | |
|---|---|
| **Transformation Name** | Edit column with formula |
| **Parameter: Columns** | myArray1_0~myArray1_8 |
| **Parameter: Formula** | IF(RAND() > 0.5, $col + (5 * RAND()), $col - RAND()) |

To make the numbers easier to manipulate, you can round them to two decimal places:

| | |
|---|---|
| **Transformation Name** | Edit column with formula |
| **Parameter: Columns** | myArray1_0~myArray1_8 |
| **Parameter: Formula** | ROUND($col, 2) |

Renest these columns into an array:

| | |
|---|---|
| **Transformation Name** | Nest columns into Objects |
| **Parameter: Columns** | myArray1_0, myArray1_1, myArray1_2, myArray1_3, myArray1_4, myArray1_5, myArray1_6, myArray1_7, myArray1_8 |
| **Parameter: Nest columns to** | Array |
| **Parameter: New column name** | 'myArray2' |

Delete the unused columns:

| | |
|---|---|
| **Transformation Name** | Delete columns |
| **Parameter: Columns** | myArray1_0~myArray1_8,myArray1 |
| **Parameter: Action** | Delete selected columns |

Your data should look similar to the following:

| myArray2 |
|---|
| ["8.29","9.63","14.63","19.63","24.63","29.63","34.63","39.63","44.63"] |
| ["8.32","14.01","19.01","24.01","29.01","34.01","39.01","44.01","49.01"] |
| ["4.55","9.58","14.58","19.58","24.58","29.58","34.58","39.58","44.58"] |
| ["9.22","14.84","19.84","24.84","29.84","34.84","39.84","44.84","49.84"] |
| ["8.75","13.36","18.36","23.36","28.36","33.36","38.36","43.36","48.36"] |
| ["8.47","14.76","19.76","24.76","29.76","34.76","39.76","44.76","49.76"] |
| ["4.93","9.99","14.99","19.99","24.99","29.99","34.99","39.99","44.99"] |

| | |
|---|---|
| ["4.65","14.98","19.98","24.98","29.98","34.98","39.98","44.98","49.98"] | |
| ["7.80","14.62","19.62","24.62","29.62","34.62","39.62","44.62","49.62"] | |
| ["9.32","9.96","14.96","19.96","24.96","29.96","34.96","39.96","44.96"] | |

**Transformation:**

These steps demonstrate the individual math functions that you can apply to your list data without unnesting it:

> **NOTE:** The NUMFORMAT function has been wrapped around each list function to account for any floating-point errors or additional digits in the results.

Sum of all values in the array (list):

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | NUMFORMAT(LISTSUM(myArray2), '#.##') |
| Parameter: New column name | 'arraySum' |

Minimum of all values in the array (list):

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | NUMFORMAT(LISTMIN(myArray2), '#.##') |
| Parameter: New column name | 'arrayMin' |

Maximum of all values in the array (list):

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | NUMFORMAT(LISTMAX(myArray2), '#.##') |
| Parameter: New column name | 'arrayMax' |

Average of all values in the array (list):

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | NUMFORMAT(LISTAVERAGE(myArray2), '#.##') |
| Parameter: New column name | 'arrayAvg' |

Variance of all values in the array (list):

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | NUMFORMAT(LISTVAR(myArray2), '#.##') |
| Parameter: New column name | 'arrayVar' |

Standard deviation of all values in the array (list):

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | NUMFORMAT(LISTSTDEV(myArray2), '#.##') |
| Parameter: New column name | 'arrayStDv' |

Mode (most common value) of all values in the array (list):

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | NUMFORMAT(LISTMODE(myArray2), '#.##') |
| Parameter: New column name | 'arrayMode' |

**Results:**

Results for the first four math functions:

| myArray2 | arrayAvg | arrayMax | arrayMin | arraySum |
|---|---|---|---|---|
| ["8.29","9.63","14.63","19.63","24.63","29.63","34.63","39.63","44.63"] | 25.04 | 44.63 | 8.29 | 225.33 |
| ["8.32","14.01","19.01","24.01","29.01","34.01","39.01","44.01","49.01"] | 28.93 | 49.01 | 8.32 | 260.4 |
| ["4.55","9.58","14.58","19.58","24.58","29.58","34.58","39.58","44.58"] | 24.58 | 44.58 | 4.55 | 221.19 |
| ["9.22","14.84","19.84","24.84","29.84","34.84","39.84","44.84","49.84"] | 29.77 | 49.84 | 9.22 | 267.94 |
| ["8.75","13.36","18.36","23.36","28.36","33.36","38.36","43.36","48.36"] | 28.4 | 48.36 | 8.75 | 255.63 |
| ["8.47","14.76","19.76","24.76","29.76","34.76","39.76","44.76","49.76"] | 29.62 | 49.76 | 8.47 | 266.55 |
| ["4.93","9.99","14.99","19.99","24.99","29.99","34.99","39.99","44.99"] | 24.98 | 44.99 | 4.93 | 224.85 |
| ["4.65","14.98","19.98","24.98","29.98","34.98","39.98","44.98","49.98"] | 29.39 | 49.98 | 4.65 | 264.49 |
| ["7.80","14.62","19.62","24.62","29.62","34.62","39.62","44.62","49.62"] | 29.42 | 49.62 | 7.8 | 264.76 |
| ["9.32","9.96","14.96","19.96","24.96","29.96","34.96","39.96","44.96"] | 25.44 | 44.96 | 9.32 | 229 |

Results for the statistical functions:

| myArray2 | arrayMode | arrayStDv | arrayVar |
|---|---|---|---|
| | | | |

| | | |
|---|---|---|
| ["8.29","9.63","14.63","19.63","24.63","29.63","34.63","39.63","44.63"] | 12.32 | 151.72 |
| ["8.32","14.01","19.01","24.01","29.01","34.01","39.01","44.01","49.01"] | 13.03 | 169.78 |
| ["4.55","9.58","14.58","19.58","24.58","29.58","34.58","39.58","44.58"] | 12.92 | 166.8 |
| ["9.22","14.84","19.84","24.84","29.84","34.84","39.84","44.84","49.84"] | 13.02 | 169.46 |
| ["8.75","13.36","18.36","23.36","28.36","33.36","38.36","43.36","48.36"] | 12.84 | 164.95 |
| ["8.47","14.76","19.76","24.76","29.76","34.76","39.76","44.76","49.76"] | 13.14 | 172.56 |
| ["4.93","9.99","14.99","19.99","24.99","29.99","34.99","39.99","44.99"] | 12.92 | 166.93 |
| ["4.65","14.98","19.98","24.98","29.98","34.98","39.98","44.98","49.98"] | 13.9 | 193.16 |
| ["7.80","14.62","19.62","24.62","29.62","34.62","39.62","44.62","49.62"] | 13.23 | 175.08 |
| ["9.32","9.96","14.96","19.96","24.96","29.96","34.96","39.96","44.96"] | 12.21 | 149.17 |

Since all values are unique within an individual array, there is no most common value in any of them, which yields empty values for the `arrayMode` column.

# LISTMIN Function

Computes the minimum of all numeric values found in input array. Input can be an array literal, a column of arrays, or a function returning an array. Input values must be of Integer or Decimal type.

When this function is invoked, all of the values in the input array are passed to the corresponding columnar function. Some restrictions may apply. See *MIN Function*.

**Wrangle vs. SQL:** This function is part of Wrangle , a proprietary data transformation language. Wrangle is not SQL. For more information, see *Wrangle Language*.

## Basic Usage

**Literal example:**

```
listmin([0,0,2,4,6,8,10,12,14,16,18,20])
```

**Output:** Returns the minimum of all values in the literal array: 0.

**Column example:**

```
listmin(myArray)
```

**Output:** Returns the minimum of all values in the arrays of the myArray column.

## Syntax and Arguments

```
listmin(array_ref)
```

| Argument | Required? | Data Type | Description |
|----------|-----------|-----------|-------------|
| array_ref | Y | Array | Array literal, reference to column containing arrays, or function returning an array |

For more information on syntax standards, see *Language Documentation Syntax Notes*.

**array_ref**

Reference to an array can be an array literal, function returning an array, or a single column containing arrays.

- If the input is not a valid numeric array, null values are returned.
- Non-numerical values within an input array are not factored in the computation.
- Multiple columns and wildcards are not supported.

**Usage Notes:**

| Required? | Data Type | Example Value |
|-----------|-----------|---------------|
| Yes | Array | myArray |

## Examples

### Example - Math functions for lists (arrays)

This example describes how to generate random array (list) data and then to apply statistical functions specifically created for arrays.

**Functions:**

| Item | Description |
|------|-------------|
| LISTSUM Function | Computes the sum of all numeric values found in input array. Input can be an array literal, a column of arrays, or a function returning an array. Input values must be of Integer or Decimal type. |
| LISTMIN Function | Computes the minimum of all numeric values found in input array. Input can be an array literal, a column of arrays, or a function returning an array. Input values must be of Integer or Decimal type. |
| LISTMAX Function | Computes the maximum of all numeric values found in input array. Input can be an array literal, a column of arrays, or a function returning an array. Input values must be of Integer or Decimal type. |
| LISTAVERAGE Function | Computes the average of all numeric values found in input array. Input can be an array literal, a column of arrays, or a function returning an array. Input values must be of Integer or Decimal type. |
| LISTVAR Function | Computes the variance of all numeric values found in input array. Input can be an array literal, a column of arrays, or a function returning an array. Input values must be of Integer or Decimal type. |
| LISTSTDEV Function | Computes the standard deviation of all numeric values found in input array. Input can be an array literal, a column of arrays, or a function returning an array. Input values must be of Integer or Decimal type. |
| LISTMODE Function | Computes the most common value of all numeric values found in input array. Input can be an array literal, a column of arrays, or a function returning an array. Input values must be of Integer or Decimal type. |

Also:

| Item | Description |
|------|-------------|
| RANGE Function | Computes an array of integers, from a beginning integer to an end (stop) integer, stepping by a third parameter. |
| RAND Function | The RAND function generates a random real number between 0 and 1. The function accepts an optional integer parameter, which causes the same set of random numbers to be generated with each job execution. |
| ROUND Function | Rounds input value to the nearest integer. Input can be an Integer, a Decimal, a column reference, or an expression. Optional second argument can be used to specify the number of digits to which to round. |

**Source:**

For this example, you can generate some randomized data using the following steps. First, you need to seed an array with a range of values using the RANGE function:

| Transformation Name | New formula |
|---------------------|-------------|
| **Parameter: Formula type** | Single row formula |
| **Parameter: Formula** | RANGE(5, 50, 5) |
| **Parameter: New column name** | 'myArray1' |

Then, unpack this array, so you can add a random factor:

| Transformation Name | Unnest Objects into columns |
|---------------------|------------------------------|
| **Parameter: Column** | myArray1 |

| Parameter: Paths to elements | '[0]', '[1]', '[2]', '[3]', '[4]', '[5]', '[6]', '[7]', '[8]', '[9]' |
|---|---|
| Parameter: Remove elements from original | true |
| Parameter: Include original column name | true |

Add the randomizing factor. Here, you are adding randomization around individual values: x-1 < x < x+4.

| Transformation Name | Edit column with formula |
|---|---|
| Parameter: Columns | myArray1_0~myArray1_8 |
| Parameter: Formula | IF(RAND() > 0.5, $col + (5 * RAND()), $col - RAND()) |

To make the numbers easier to manipulate, you can round them to two decimal places:

| Transformation Name | Edit column with formula |
|---|---|
| Parameter: Columns | myArray1_0~myArray1_8 |
| Parameter: Formula | ROUND($col, 2) |

Renest these columns into an array:

| Transformation Name | Nest columns into Objects |
|---|---|
| Parameter: Columns | myArray1_0, myArray1_1, myArray1_2, myArray1_3, myArray1_4, myArray1_5, myArray1_6, myArray1_7, myArray1_8 |
| Parameter: Nest columns to | Array |
| Parameter: New column name | 'myArray2' |

Delete the unused columns:

| Transformation Name | Delete columns |
|---|---|
| Parameter: Columns | myArray1_0~myArray1_8,myArray1 |
| Parameter: Action | Delete selected columns |

Your data should look similar to the following:

| myArray2 |
|---|
| ["8.29","9.63","14.63","19.63","24.63","29.63","34.63","39.63","44.63"] |
| ["8.32","14.01","19.01","24.01","29.01","34.01","39.01","44.01","49.01"] |
| ["4.55","9.58","14.58","19.58","24.58","29.58","34.58","39.58","44.58"] |
| ["9.22","14.84","19.84","24.84","29.84","34.84","39.84","44.84","49.84"] |
| ["8.75","13.36","18.36","23.36","28.36","33.36","38.36","43.36","48.36"] |
| ["8.47","14.76","19.76","24.76","29.76","34.76","39.76","44.76","49.76"] |

| ["4.93","9.99","14.99","19.99","24.99","29.99","34.99","39.99","44.99"] |
| --- |
| ["4.65","14.98","19.98","24.98","29.98","34.98","39.98","44.98","49.98"] |
| ["7.80","14.62","19.62","24.62","29.62","34.62","39.62","44.62","49.62"] |
| ["9.32","9.96","14.96","19.96","24.96","29.96","34.96","39.96","44.96"] |

**Transformation:**

These steps demonstrate the individual math functions that you can apply to your list data without unnesting it:

> **NOTE:** The NUMFORMAT function has been wrapped around each list function to account for any floating-point errors or additional digits in the results.

Sum of all values in the array (list):

| Transformation Name | New formula |
| --- | --- |
| Parameter: Formula type | Single row formula |
| Parameter: Formula | NUMFORMAT(LISTSUM(myArray2), '#.##') |
| Parameter: New column name | 'arraySum' |

Minimum of all values in the array (list):

| Transformation Name | New formula |
| --- | --- |
| Parameter: Formula type | Single row formula |
| Parameter: Formula | NUMFORMAT(LISTMIN(myArray2), '#.##') |
| Parameter: New column name | 'arrayMin' |

Maximum of all values in the array (list):

| Transformation Name | New formula |
| --- | --- |
| Parameter: Formula type | Single row formula |
| Parameter: Formula | NUMFORMAT(LISTMAX(myArray2), '#.##') |
| Parameter: New column name | 'arrayMax' |

Average of all values in the array (list):

| Transformation Name | New formula |
| --- | --- |
| Parameter: Formula type | Single row formula |
| Parameter: Formula | NUMFORMAT(LISTAVERAGE(myArray2), '#.##') |
| Parameter: New column name | 'arrayAvg' |

Variance of all values in the array (list):

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | NUMFORMAT(LISTVAR(myArray2), '#.##') |
| Parameter: New column name | 'arrayVar' |

Standard deviation of all values in the array (list):

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | NUMFORMAT(LISTSTDEV(myArray2), '#.##') |
| Parameter: New column name | 'arrayStDv' |

Mode (most common value) of all values in the array (list):

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | NUMFORMAT(LISTMODE(myArray2), '#.##') |
| Parameter: New column name | 'arrayMode' |

**Results:**

Results for the first four math functions:

| myArray2 | arrayAvg | arrayMax | arrayMin | arraySum |
|---|---|---|---|---|
| ["8.29","9.63","14.63","19.63","24.63","29.63","34.63","39.63","44.63"] | 25.04 | 44.63 | 8.29 | 225.33 |
| ["8.32","14.01","19.01","24.01","29.01","34.01","39.01","44.01","49.01"] | 28.93 | 49.01 | 8.32 | 260.4 |
| ["4.55","9.58","14.58","19.58","24.58","29.58","34.58","39.58","44.58"] | 24.58 | 44.58 | 4.55 | 221.19 |
| ["9.22","14.84","19.84","24.84","29.84","34.84","39.84","44.84","49.84"] | 29.77 | 49.84 | 9.22 | 267.94 |
| ["8.75","13.36","18.36","23.36","28.36","33.36","38.36","43.36","48.36"] | 28.4 | 48.36 | 8.75 | 255.63 |
| ["8.47","14.76","19.76","24.76","29.76","34.76","39.76","44.76","49.76"] | 29.62 | 49.76 | 8.47 | 266.55 |
| ["4.93","9.99","14.99","19.99","24.99","29.99","34.99","39.99","44.99"] | 24.98 | 44.99 | 4.93 | 224.85 |
| ["4.65","14.98","19.98","24.98","29.98","34.98","39.98","44.98","49.98"] | 29.39 | 49.98 | 4.65 | 264.49 |
| ["7.80","14.62","19.62","24.62","29.62","34.62","39.62","44.62","49.62"] | 29.42 | 49.62 | 7.8 | 264.76 |
| ["9.32","9.96","14.96","19.96","24.96","29.96","34.96","39.96","44.96"] | 25.44 | 44.96 | 9.32 | 229 |

Results for the statistical functions:

| myArray2 | arrayMode | arrayStDv | arrayVar |
|---|---|---|---|

| | arrayMode | | |
| --- | --- | --- | --- |
| ["8.29","9.63","14.63","19.63","24.63","29.63","34.63","39.63","44.63"] | | 12.32 | 151.72 |
| ["8.32","14.01","19.01","24.01","29.01","34.01","39.01","44.01","49.01"] | | 13.03 | 169.78 |
| ["4.55","9.58","14.58","19.58","24.58","29.58","34.58","39.58","44.58"] | | 12.92 | 166.8 |
| ["9.22","14.84","19.84","24.84","29.84","34.84","39.84","44.84","49.84"] | | 13.02 | 169.46 |
| ["8.75","13.36","18.36","23.36","28.36","33.36","38.36","43.36","48.36"] | | 12.84 | 164.95 |
| ["8.47","14.76","19.76","24.76","29.76","34.76","39.76","44.76","49.76"] | | 13.14 | 172.56 |
| ["4.93","9.99","14.99","19.99","24.99","29.99","34.99","39.99","44.99"] | | 12.92 | 166.93 |
| ["4.65","14.98","19.98","24.98","29.98","34.98","39.98","44.98","49.98"] | | 13.9 | 193.16 |
| ["7.80","14.62","19.62","24.62","29.62","34.62","39.62","44.62","49.62"] | | 13.23 | 175.08 |
| ["9.32","9.96","14.96","19.96","24.96","29.96","34.96","39.96","44.96"] | | 12.21 | 149.17 |

Since all values are unique within an individual array, there is no most common value in any of them, which yields empty values for the `arrayMode` column.

# LISTMODE Function

Computes the most common value of all numeric values found in input array. Input can be an array literal, a column of arrays, or a function returning an array. Input values must be of Integer or Decimal type.

When this function is invoked, all of the values in the input array are passed to the corresponding columnar function. Some restrictions may apply. See *MODE Function*.

**Wrangle vs. SQL:** This function is part of Wrangle , a proprietary data transformation language. Wrangle is not SQL. For more information, see *Wrangle Language*.

## Basic Usage

**Literal example:**

```
listmode([0,0,2,4,6,8,10,12,14,16,18,20])
```

**Output:** Returns most common of all values in the literal array: `0`.

**Column example:**

```
listmode(myArray)
```

**Output:** Generates an output column containing the mode of all values in the arrays of the `myArray` column.

## Syntax and Arguments

```
listmode(array_ref)
```

| Argument | Required? | Data Type | Description |
|----------|-----------|-----------|-------------|
| array_ref | Y | Array | Array literal, reference to column containing arrays, or function returning an array |

For more information on syntax standards, see *Language Documentation Syntax Notes*.

**array_ref**

Reference to an array can be an array literal, function returning an array, or a single column containing arrays.

- If the input is not a valid numeric array, null values are returned.
- Non-numerical values within an input array are not factored in the computation.
- Multiple columns and wildcards are not supported.

**Usage Notes:**

| Required? | Data Type | Example Value |
|-----------|-----------|---------------|
| Yes | Array | `myArray` |

## Examples

> **Tip:** For additional examples, see *Common Tasks*.

## Example - Math functions for lists (arrays)

This example describes how to generate random array (list) data and then to apply statistical functions specifically created for arrays.

**Functions:**

| Item | Description |
|------|-------------|
| LISTSUM Function | Computes the sum of all numeric values found in input array. Input can be an array literal, a column of arrays, or a function returning an array. Input values must be of Integer or Decimal type. |
| LISTMIN Function | Computes the minimum of all numeric values found in input array. Input can be an array literal, a column of arrays, or a function returning an array. Input values must be of Integer or Decimal type. |
| LISTMAX Function | Computes the maximum of all numeric values found in input array. Input can be an array literal, a column of arrays, or a function returning an array. Input values must be of Integer or Decimal type. |
| LISTAVERAGE Function | Computes the average of all numeric values found in input array. Input can be an array literal, a column of arrays, or a function returning an array. Input values must be of Integer or Decimal type. |
| LISTVAR Function | Computes the variance of all numeric values found in input array. Input can be an array literal, a column of arrays, or a function returning an array. Input values must be of Integer or Decimal type. |
| LISTSTDEV Function | Computes the standard deviation of all numeric values found in input array. Input can be an array literal, a column of arrays, or a function returning an array. Input values must be of Integer or Decimal type. |
| LISTMODE Function | Computes the most common value of all numeric values found in input array. Input can be an array literal, a column of arrays, or a function returning an array. Input values must be of Integer or Decimal type. |

Also:

| Item | Description |
|------|-------------|
| RANGE Function | Computes an array of integers, from a beginning integer to an end (stop) integer, stepping by a third parameter. |
| RAND Function | The RAND function generates a random real number between 0 and 1. The function accepts an optional integer parameter, which causes the same set of random numbers to be generated with each job execution. |
| ROUND Function | Rounds input value to the nearest integer. Input can be an Integer, a Decimal, a column reference, or an expression. Optional second argument can be used to specify the number of digits to which to round. |

**Source:**

For this example, you can generate some randomized data using the following steps. First, you need to seed an array with a range of values using the RANGE function:

| | |
|------|------|
| **Transformation Name** | New formula |
| **Parameter: Formula type** | Single row formula |
| **Parameter: Formula** | RANGE(5, 50, 5) |
| **Parameter: New column name** | 'myArray1' |

Then, unpack this array, so you can add a random factor:

| | |
|------|------|
| **Transformation Name** | Unnest Objects into columns |
| **Parameter: Column** | myArray1 |
| **Parameter: Paths to elements** | '[0]', '[1]', '[2]', '[3]', '[4]', '[5]', '[6]', |

| | '[7]', '[8]', '[9]' |
|---|---|
| **Parameter: Remove elements from original** | `true` |
| **Parameter: Include original column name** | `true` |

Add the randomizing factor. Here, you are adding randomization around individual values: x-1 < x < x+4.

| | |
|---|---|
| **Transformation Name** | `Edit column with formula` |
| **Parameter: Columns** | `myArray1_0~myArray1_8` |
| **Parameter: Formula** | `IF(RAND() > 0.5, $col + (5 * RAND()), $col - RAND())` |

To make the numbers easier to manipulate, you can round them to two decimal places:

| | |
|---|---|
| **Transformation Name** | `Edit column with formula` |
| **Parameter: Columns** | `myArray1_0~myArray1_8` |
| **Parameter: Formula** | `ROUND($col, 2)` |

Renest these columns into an array:

| | |
|---|---|
| **Transformation Name** | `Nest columns into Objects` |
| **Parameter: Columns** | `myArray1_0, myArray1_1, myArray1_2, myArray1_3, myArray1_4, myArray1_5, myArray1_6, myArray1_7, myArray1_8` |
| **Parameter: Nest columns to** | `Array` |
| **Parameter: New column name** | `'myArray2'` |

Delete the unused columns:

| | |
|---|---|
| **Transformation Name** | `Delete columns` |
| **Parameter: Columns** | `myArray1_0~myArray1_8,myArray1` |
| **Parameter: Action** | `Delete selected columns` |

Your data should look similar to the following:

| myArray2 |
|---|
| ["8.29","9.63","14.63","19.63","24.63","29.63","34.63","39.63","44.63"] |
| ["8.32","14.01","19.01","24.01","29.01","34.01","39.01","44.01","49.01"] |
| ["4.55","9.58","14.58","19.58","24.58","29.58","34.58","39.58","44.58"] |
| ["9.22","14.84","19.84","24.84","29.84","34.84","39.84","44.84","49.84"] |
| ["8.75","13.36","18.36","23.36","28.36","33.36","38.36","43.36","48.36"] |
| ["8.47","14.76","19.76","24.76","29.76","34.76","39.76","44.76","49.76"] |
| ["4.93","9.99","14.99","19.99","24.99","29.99","34.99","39.99","44.99"] |

| |
|---|
| ["4.65","14.98","19.98","24.98","29.98","34.98","39.98","44.98","49.98"] |
| ["7.80","14.62","19.62","24.62","29.62","34.62","39.62","44.62","49.62"] |
| ["9.32","9.96","14.96","19.96","24.96","29.96","34.96","39.96","44.96"] |

**Transformation:**

These steps demonstrate the individual math functions that you can apply to your list data without unnesting it:

> **NOTE:** The NUMFORMAT function has been wrapped around each list function to account for any floating-point errors or additional digits in the results.

Sum of all values in the array (list):

| | |
|---|---|
| **Transformation Name** | New formula |
| **Parameter: Formula type** | Single row formula |
| **Parameter: Formula** | NUMFORMAT(LISTSUM(myArray2), '#.##') |
| **Parameter: New column name** | 'arraySum' |

Minimum of all values in the array (list):

| | |
|---|---|
| **Transformation Name** | New formula |
| **Parameter: Formula type** | Single row formula |
| **Parameter: Formula** | NUMFORMAT(LISTMIN(myArray2), '#.##') |
| **Parameter: New column name** | 'arrayMin' |

Maximum of all values in the array (list):

| | |
|---|---|
| **Transformation Name** | New formula |
| **Parameter: Formula type** | Single row formula |
| **Parameter: Formula** | NUMFORMAT(LISTMAX(myArray2), '#.##') |
| **Parameter: New column name** | 'arrayMax' |

Average of all values in the array (list):

| | |
|---|---|
| **Transformation Name** | New formula |
| **Parameter: Formula type** | Single row formula |
| **Parameter: Formula** | NUMFORMAT(LISTAVERAGE(myArray2), '#.##') |
| **Parameter: New column name** | 'arrayAvg' |

Variance of all values in the array (list):

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | NUMFORMAT(LISTVAR(myArray2), '#.##') |
| Parameter: New column name | 'arrayVar' |

Standard deviation of all values in the array (list):

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | NUMFORMAT(LISTSTDEV(myArray2), '#.##') |
| Parameter: New column name | 'arrayStDv' |

Mode (most common value) of all values in the array (list):

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | NUMFORMAT(LISTMODE(myArray2), '#.##') |
| Parameter: New column name | 'arrayMode' |

**Results:**

Results for the first four math functions:

| myArray2 | arrayAvg | arrayMax | arrayMin | arraySum |
|---|---|---|---|---|
| ["8.29","9.63","14.63","19.63","24.63","29.63","34.63","39.63","44.63"] | 25.04 | 44.63 | 8.29 | 225.33 |
| ["8.32","14.01","19.01","24.01","29.01","34.01","39.01","44.01","49.01"] | 28.93 | 49.01 | 8.32 | 260.4 |
| ["4.55","9.58","14.58","19.58","24.58","29.58","34.58","39.58","44.58"] | 24.58 | 44.58 | 4.55 | 221.19 |
| ["9.22","14.84","19.84","24.84","29.84","34.84","39.84","44.84","49.84"] | 29.77 | 49.84 | 9.22 | 267.94 |
| ["8.75","13.36","18.36","23.36","28.36","33.36","38.36","43.36","48.36"] | 28.4 | 48.36 | 8.75 | 255.63 |
| ["8.47","14.76","19.76","24.76","29.76","34.76","39.76","44.76","49.76"] | 29.62 | 49.76 | 8.47 | 266.55 |
| ["4.93","9.99","14.99","19.99","24.99","29.99","34.99","39.99","44.99"] | 24.98 | 44.99 | 4.93 | 224.85 |
| ["4.65","14.98","19.98","24.98","29.98","34.98","39.98","44.98","49.98"] | 29.39 | 49.98 | 4.65 | 264.49 |
| ["7.80","14.62","19.62","24.62","29.62","34.62","39.62","44.62","49.62"] | 29.42 | 49.62 | 7.8 | 264.76 |
| ["9.32","9.96","14.96","19.96","24.96","29.96","34.96","39.96","44.96"] | 25.44 | 44.96 | 9.32 | 229 |

Results for the statistical functions:

| myArray2 | arrayMode | arrayStDv | arrayVar |
|---|---|---|---|
| | | | |

| | | |
|---|---|---|
| ["8.29","9.63","14.63","19.63","24.63","29.63","34.63","39.63","44.63"] | 12.32 | 151.72 |
| ["8.32","14.01","19.01","24.01","29.01","34.01","39.01","44.01","49.01"] | 13.03 | 169.78 |
| ["4.55","9.58","14.58","19.58","24.58","29.58","34.58","39.58","44.58"] | 12.92 | 166.8 |
| ["9.22","14.84","19.84","24.84","29.84","34.84","39.84","44.84","49.84"] | 13.02 | 169.46 |
| ["8.75","13.36","18.36","23.36","28.36","33.36","38.36","43.36","48.36"] | 12.84 | 164.95 |
| ["8.47","14.76","19.76","24.76","29.76","34.76","39.76","44.76","49.76"] | 13.14 | 172.56 |
| ["4.93","9.99","14.99","19.99","24.99","29.99","34.99","39.99","44.99"] | 12.92 | 166.93 |
| ["4.65","14.98","19.98","24.98","29.98","34.98","39.98","44.98","49.98"] | 13.9 | 193.16 |
| ["7.80","14.62","19.62","24.62","29.62","34.62","39.62","44.62","49.62"] | 13.23 | 175.08 |
| ["9.32","9.96","14.96","19.96","24.96","29.96","34.96","39.96","44.96"] | 12.21 | 149.17 |

Since all values are unique within an individual array, there is no most common value in any of them, which yields empty values for the `arrayMode` column.

# LISTSTDEV Function

Computes the standard deviation of all numeric values found in input array. Input can be an array literal, a column of arrays, or a function returning an array. Input values must be of Integer or Decimal type.

When this function is invoked, all of the values in the input array are passed to the corresponding columnar function. Some restrictions may apply. See *STDEV Function*.

**Wrangle vs. SQL:** This function is part of Wrangle , a proprietary data transformation language. Wrangle is not SQL. For more information, see *Wrangle Language*.

## Basic Usage

**Literal example:**

```
liststdev([0,0,2,4,6,8,10,12,14,16,18,20])
```

**Output:** Returns the standard deviation of all values in the literal array: `6.952217872`.

**Column example:**

```
liststdev(myArray)
```

**Output:** Generates an output column containing the standard deviation of all values in the arrays of the `myArray` column.

## Syntax and Arguments

```
liststdev(array_ref)
```

| Argument | Required? | Data Type | Description |
|----------|-----------|-----------|-------------|
| array_ref | Y | Array | Array literal, reference to column containing arrays, or function returning an array |

For more information on syntax standards, see *Language Documentation Syntax Notes*.

**array_ref**

Reference to an array can be an array literal, function returning an array, or a single column containing arrays.

- If the input is not a valid numeric array, null values are returned.
- Non-numerical values within an input array are not factored in the computation.
- Multiple columns and wildcards are not supported.

**Usage Notes:**

| Required? | Data Type | Example Value |
|-----------|-----------|---------------|
| Yes | Array | `myArray` |

## Examples

### Example - Math functions for lists (arrays)

This example describes how to generate random array (list) data and then to apply statistical functions specifically created for arrays.

**Functions:**

| Item | Description |
|---|---|
| LISTSUM Function | Computes the sum of all numeric values found in input array. Input can be an array literal, a column of arrays, or a function returning an array. Input values must be of Integer or Decimal type. |
| LISTMIN Function | Computes the minimum of all numeric values found in input array. Input can be an array literal, a column of arrays, or a function returning an array. Input values must be of Integer or Decimal type. |
| LISTMAX Function | Computes the maximum of all numeric values found in input array. Input can be an array literal, a column of arrays, or a function returning an array. Input values must be of Integer or Decimal type. |
| LISTAVERAGE Function | Computes the average of all numeric values found in input array. Input can be an array literal, a column of arrays, or a function returning an array. Input values must be of Integer or Decimal type. |
| LISTVAR Function | Computes the variance of all numeric values found in input array. Input can be an array literal, a column of arrays, or a function returning an array. Input values must be of Integer or Decimal type. |
| LISTSTDEV Function | Computes the standard deviation of all numeric values found in input array. Input can be an array literal, a column of arrays, or a function returning an array. Input values must be of Integer or Decimal type. |
| LISTMODE Function | Computes the most common value of all numeric values found in input array. Input can be an array literal, a column of arrays, or a function returning an array. Input values must be of Integer or Decimal type. |

Also:

| Item | Description |
|---|---|
| RANGE Function | Computes an array of integers, from a beginning integer to an end (stop) integer, stepping by a third parameter. |
| RAND Function | The RAND function generates a random real number between 0 and 1. The function accepts an optional integer parameter, which causes the same set of random numbers to be generated with each job execution. |
| ROUND Function | Rounds input value to the nearest integer. Input can be an Integer, a Decimal, a column reference, or an expression. Optional second argument can be used to specify the number of digits to which to round. |

**Source:**

For this example, you can generate some randomized data using the following steps. First, you need to seed an array with a range of values using the RANGE function:

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | RANGE(5, 50, 5) |
| Parameter: New column name | 'myArray1' |

Then, unpack this array, so you can add a random factor:

| Transformation Name | Unnest Objects into columns |
|---|---|
| Parameter: Column | myArray1 |

| Parameter: Paths to elements | `'[0]', '[1]', '[2]', '[3]', '[4]', '[5]', '[6]', '[7]', '[8]', '[9]'` |
|---|---|
| Parameter: Remove elements from original | `true` |
| Parameter: Include original column name | `true` |

Add the randomizing factor. Here, you are adding randomization around individual values:  x-1 < x < x+4.

| Transformation Name | `Edit column with formula` |
|---|---|
| Parameter: Columns | `myArray1_0~myArray1_8` |
| Parameter: Formula | `IF(RAND() > 0.5, $col + (5 * RAND()), $col - RAND())` |

To make the numbers easier to manipulate, you can round them to two decimal places:

| Transformation Name | `Edit column with formula` |
|---|---|
| Parameter: Columns | `myArray1_0~myArray1_8` |
| Parameter: Formula | `ROUND($col, 2)` |

Renest these columns into an array:

| Transformation Name | `Nest columns into Objects` |
|---|---|
| Parameter: Columns | `myArray1_0, myArray1_1, myArray1_2, myArray1_3, myArray1_4, myArray1_5, myArray1_6, myArray1_7, myArray1_8` |
| Parameter: Nest columns to | `Array` |
| Parameter: New column name | `'myArray2'` |

Delete the unused columns:

| Transformation Name | `Delete columns` |
|---|---|
| Parameter: Columns | `myArray1_0~myArray1_8,myArray1` |
| Parameter: Action | `Delete selected columns` |

Your data should look similar to the following:

| myArray2 |
|---|
| ["8.29","9.63","14.63","19.63","24.63","29.63","34.63","39.63","44.63"] |
| ["8.32","14.01","19.01","24.01","29.01","34.01","39.01","44.01","49.01"] |
| ["4.55","9.58","14.58","19.58","24.58","29.58","34.58","39.58","44.58"] |
| ["9.22","14.84","19.84","24.84","29.84","34.84","39.84","44.84","49.84"] |
| ["8.75","13.36","18.36","23.36","28.36","33.36","38.36","43.36","48.36"] |
| ["8.47","14.76","19.76","24.76","29.76","34.76","39.76","44.76","49.76"] |

| | |
|---|---|
| ["4.93","9.99","14.99","19.99","24.99","29.99","34.99","39.99","44.99"] | |
| ["4.65","14.98","19.98","24.98","29.98","34.98","39.98","44.98","49.98"] | |
| ["7.80","14.62","19.62","24.62","29.62","34.62","39.62","44.62","49.62"] | |
| ["9.32","9.96","14.96","19.96","24.96","29.96","34.96","39.96","44.96"] | |

**Transformation:**

These steps demonstrate the individual math functions that you can apply to your list data without unnesting it:

> **NOTE:** The NUMFORMAT function has been wrapped around each list function to account for any floating-point errors or additional digits in the results.

Sum of all values in the array (list):

| | |
|---|---|
| **Transformation Name** | New formula |
| **Parameter: Formula type** | Single row formula |
| **Parameter: Formula** | NUMFORMAT(LISTSUM(myArray2), '#.##') |
| **Parameter: New column name** | 'arraySum' |

Minimum of all values in the array (list):

| | |
|---|---|
| **Transformation Name** | New formula |
| **Parameter: Formula type** | Single row formula |
| **Parameter: Formula** | NUMFORMAT(LISTMIN(myArray2), '#.##') |
| **Parameter: New column name** | 'arrayMin' |

Maximum of all values in the array (list):

| | |
|---|---|
| **Transformation Name** | New formula |
| **Parameter: Formula type** | Single row formula |
| **Parameter: Formula** | NUMFORMAT(LISTMAX(myArray2), '#.##') |
| **Parameter: New column name** | 'arrayMax' |

Average of all values in the array (list):

| | |
|---|---|
| **Transformation Name** | New formula |
| **Parameter: Formula type** | Single row formula |
| **Parameter: Formula** | NUMFORMAT(LISTAVERAGE(myArray2), '#.##') |
| **Parameter: New column name** | 'arrayAvg' |

Variance of all values in the array (list):

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | NUMFORMAT(LISTVAR(myArray2), '#.##') |
| Parameter: New column name | 'arrayVar' |

Standard deviation of all values in the array (list):

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | NUMFORMAT(LISTSTDEV(myArray2), '#.##') |
| Parameter: New column name | 'arrayStDv' |

Mode (most common value) of all values in the array (list):

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | NUMFORMAT(LISTMODE(myArray2), '#.##') |
| Parameter: New column name | 'arrayMode' |

**Results:**

Results for the first four math functions:

| myArray2 | arrayAvg | arrayMax | arrayMin | arraySum |
|---|---|---|---|---|
| ["8.29","9.63","14.63","19.63","24.63","29.63","34.63","39.63","44.63"] | 25.04 | 44.63 | 8.29 | 225.33 |
| ["8.32","14.01","19.01","24.01","29.01","34.01","39.01","44.01","49.01"] | 28.93 | 49.01 | 8.32 | 260.4 |
| ["4.55","9.58","14.58","19.58","24.58","29.58","34.58","39.58","44.58"] | 24.58 | 44.58 | 4.55 | 221.19 |
| ["9.22","14.84","19.84","24.84","29.84","34.84","39.84","44.84","49.84"] | 29.77 | 49.84 | 9.22 | 267.94 |
| ["8.75","13.36","18.36","23.36","28.36","33.36","38.36","43.36","48.36"] | 28.4 | 48.36 | 8.75 | 255.63 |
| ["8.47","14.76","19.76","24.76","29.76","34.76","39.76","44.76","49.76"] | 29.62 | 49.76 | 8.47 | 266.55 |
| ["4.93","9.99","14.99","19.99","24.99","29.99","34.99","39.99","44.99"] | 24.98 | 44.99 | 4.93 | 224.85 |
| ["4.65","14.98","19.98","24.98","29.98","34.98","39.98","44.98","49.98"] | 29.39 | 49.98 | 4.65 | 264.49 |
| ["7.80","14.62","19.62","24.62","29.62","34.62","39.62","44.62","49.62"] | 29.42 | 49.62 | 7.8 | 264.76 |
| ["9.32","9.96","14.96","19.96","24.96","29.96","34.96","39.96","44.96"] | 25.44 | 44.96 | 9.32 | 229 |

Results for the statistical functions:

| myArray2 | arrayMode | arrayStDv | arrayVar |
|---|---|---|---|
| ["8.29","9.63","14.63","19.63","24.63","29.63","34.63","39.63","44.63"] | | 12.32 | 151.72 |
| ["8.32","14.01","19.01","24.01","29.01","34.01","39.01","44.01","49.01"] | | 13.03 | 169.78 |
| ["4.55","9.58","14.58","19.58","24.58","29.58","34.58","39.58","44.58"] | | 12.92 | 166.8 |
| ["9.22","14.84","19.84","24.84","29.84","34.84","39.84","44.84","49.84"] | | 13.02 | 169.46 |
| ["8.75","13.36","18.36","23.36","28.36","33.36","38.36","43.36","48.36"] | | 12.84 | 164.95 |
| ["8.47","14.76","19.76","24.76","29.76","34.76","39.76","44.76","49.76"] | | 13.14 | 172.56 |
| ["4.93","9.99","14.99","19.99","24.99","29.99","34.99","39.99","44.99"] | | 12.92 | 166.93 |
| ["4.65","14.98","19.98","24.98","29.98","34.98","39.98","44.98","49.98"] | | 13.9 | 193.16 |
| ["7.80","14.62","19.62","24.62","29.62","34.62","39.62","44.62","49.62"] | | 13.23 | 175.08 |
| ["9.32","9.96","14.96","19.96","24.96","29.96","34.96","39.96","44.96"] | | 12.21 | 149.17 |

Since all values are unique within an individual array, there is no most common value in any of them, which yields empty values for the `arrayMode` column.

# LISTSUM Function

Computes the sum of all numeric values found in input array. Input can be an array literal, a column of arrays, or a function returning an array. Input values must be of Integer or Decimal type.

When this function is invoked, all of the values in the input array are passed to the corresponding columnar function. Some restrictions may apply. See *SUM Function*.

**Wrangle vs. SQL:** This function is part of Wrangle , a proprietary data transformation language. Wrangle is not SQL. For more information, see *Wrangle Language*.

## Basic Usage

**Literal example:**

```
listsum([0,0,2,4,6,8,10,12,14,16,18,20])
```

**Output:** Returns the sum of all values in the literal array: `110`.

**Column example:**

```
listsum(myArray)
```

**Output:** Returns the sum of all values in the arrays of the `myArray` column.

## Syntax and Arguments

```
listsum(array_ref)
```

| Argument | Required? | Data Type | Description |
|---|---|---|---|
| array_ref | Y | Array | Array literal, reference to column containing arrays, or function returning an array |

For more information on syntax standards, see *Language Documentation Syntax Notes*.

**array_ref**

Reference to an array can be an array literal, function returning an array, or a single column containing arrays.

- If the input is not a valid numeric array, null values are returned.
- Non-numerical values within an input array are not factored in the computation.
- Multiple columns and wildcards are not supported.

**Usage Notes:**

| Required? | Data Type | Example Value |
|---|---|---|
| Yes | Array | `myArray` |

## Examples

> **Tip:** For additional examples, see *Common Tasks*.

## Example - Math functions for lists (arrays)

This example describes how to generate random array (list) data and then to apply statistical functions specifically created for arrays.

**Functions:**

| Item | Description |
|------|-------------|
| LISTSUM Function | Computes the sum of all numeric values found in input array. Input can be an array literal, a column of arrays, or a function returning an array. Input values must be of Integer or Decimal type. |
| LISTMIN Function | Computes the minimum of all numeric values found in input array. Input can be an array literal, a column of arrays, or a function returning an array. Input values must be of Integer or Decimal type. |
| LISTMAX Function | Computes the maximum of all numeric values found in input array. Input can be an array literal, a column of arrays, or a function returning an array. Input values must be of Integer or Decimal type. |
| LISTAVERAGE Function | Computes the average of all numeric values found in input array. Input can be an array literal, a column of arrays, or a function returning an array. Input values must be of Integer or Decimal type. |
| LISTVAR Function | Computes the variance of all numeric values found in input array. Input can be an array literal, a column of arrays, or a function returning an array. Input values must be of Integer or Decimal type. |
| LISTSTDEV Function | Computes the standard deviation of all numeric values found in input array. Input can be an array literal, a column of arrays, or a function returning an array. Input values must be of Integer or Decimal type. |
| LISTMODE Function | Computes the most common value of all numeric values found in input array. Input can be an array literal, a column of arrays, or a function returning an array. Input values must be of Integer or Decimal type. |

Also:

| Item | Description |
|------|-------------|
| RANGE Function | Computes an array of integers, from a beginning integer to an end (stop) integer, stepping by a third parameter. |
| RAND Function | The RAND function generates a random real number between 0 and 1. The function accepts an optional integer parameter, which causes the same set of random numbers to be generated with each job execution. |
| ROUND Function | Rounds input value to the nearest integer. Input can be an Integer, a Decimal, a column reference, or an expression. Optional second argument can be used to specify the number of digits to which to round. |

**Source:**

For this example, you can generate some randomized data using the following steps. First, you need to seed an array with a range of values using the RANGE function:

| | |
|---|---|
| **Transformation Name** | New formula |
| **Parameter: Formula type** | Single row formula |
| **Parameter: Formula** | RANGE(5, 50, 5) |
| **Parameter: New column name** | 'myArray1' |

Then, unpack this array, so you can add a random factor:

| | |
|---|---|
| **Transformation Name** | Unnest Objects into columns |
| **Parameter: Column** | myArray1 |
| **Parameter: Paths to elements** | '[0]', '[1]', '[2]', '[3]', '[4]', '[5]', '[6]', |

| | '[7]', '[8]', '[9]' |
|---|---|
| **Parameter: Remove elements from original** | true |
| **Parameter: Include original column name** | true |

Add the randomizing factor. Here, you are adding randomization around individual values: x-1 < x < x+4.

| | |
|---|---|
| **Transformation Name** | Edit column with formula |
| **Parameter: Columns** | myArray1_0~myArray1_8 |
| **Parameter: Formula** | IF(RAND() > 0.5, $col + (5 * RAND()), $col - RAND()) |

To make the numbers easier to manipulate, you can round them to two decimal places:

| | |
|---|---|
| **Transformation Name** | Edit column with formula |
| **Parameter: Columns** | myArray1_0~myArray1_8 |
| **Parameter: Formula** | ROUND($col, 2) |

Renest these columns into an array:

| | |
|---|---|
| **Transformation Name** | Nest columns into Objects |
| **Parameter: Columns** | myArray1_0, myArray1_1, myArray1_2, myArray1_3, myArray1_4, myArray1_5, myArray1_6, myArray1_7, myArray1_8 |
| **Parameter: Nest columns to** | Array |
| **Parameter: New column name** | 'myArray2' |

Delete the unused columns:

| | |
|---|---|
| **Transformation Name** | Delete columns |
| **Parameter: Columns** | myArray1_0~myArray1_8,myArray1 |
| **Parameter: Action** | Delete selected columns |

Your data should look similar to the following:

| myArray2 |
|---|
| ["8.29","9.63","14.63","19.63","24.63","29.63","34.63","39.63","44.63"] |
| ["8.32","14.01","19.01","24.01","29.01","34.01","39.01","44.01","49.01"] |
| ["4.55","9.58","14.58","19.58","24.58","29.58","34.58","39.58","44.58"] |
| ["9.22","14.84","19.84","24.84","29.84","34.84","39.84","44.84","49.84"] |
| ["8.75","13.36","18.36","23.36","28.36","33.36","38.36","43.36","48.36"] |
| ["8.47","14.76","19.76","24.76","29.76","34.76","39.76","44.76","49.76"] |
| ["4.93","9.99","14.99","19.99","24.99","29.99","34.99","39.99","44.99"] |

| | |
|---|---|
| ["4.65","14.98","19.98","24.98","29.98","34.98","39.98","44.98","49.98"] | |
| ["7.80","14.62","19.62","24.62","29.62","34.62","39.62","44.62","49.62"] | |
| ["9.32","9.96","14.96","19.96","24.96","29.96","34.96","39.96","44.96"] | |

**Transformation:**

These steps demonstrate the individual math functions that you can apply to your list data without unnesting it:

> **NOTE:** The NUMFORMAT function has been wrapped around each list function to account for any floating-point errors or additional digits in the results.

Sum of all values in the array (list):

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | NUMFORMAT(LISTSUM(myArray2), '#.##') |
| Parameter: New column name | 'arraySum' |

Minimum of all values in the array (list):

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | NUMFORMAT(LISTMIN(myArray2), '#.##') |
| Parameter: New column name | 'arrayMin' |

Maximum of all values in the array (list):

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | NUMFORMAT(LISTMAX(myArray2), '#.##') |
| Parameter: New column name | 'arrayMax' |

Average of all values in the array (list):

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | NUMFORMAT(LISTAVERAGE(myArray2), '#.##') |
| Parameter: New column name | 'arrayAvg' |

Variance of all values in the array (list):

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | NUMFORMAT(LISTVAR(myArray2), '#.##') |
| Parameter: New column name | 'arrayVar' |

Standard deviation of all values in the array (list):

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | NUMFORMAT(LISTSTDEV(myArray2), '#.##') |
| Parameter: New column name | 'arrayStDv' |

Mode (most common value) of all values in the array (list):

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | NUMFORMAT(LISTMODE(myArray2), '#.##') |
| Parameter: New column name | 'arrayMode' |

**Results:**

Results for the first four math functions:

| myArray2 | arrayAvg | arrayMax | arrayMin | arraySum |
|---|---|---|---|---|
| ["8.29","9.63","14.63","19.63","24.63","29.63","34.63","39.63","44.63"] | 25.04 | 44.63 | 8.29 | 225.33 |
| ["8.32","14.01","19.01","24.01","29.01","34.01","39.01","44.01","49.01"] | 28.93 | 49.01 | 8.32 | 260.4 |
| ["4.55","9.58","14.58","19.58","24.58","29.58","34.58","39.58","44.58"] | 24.58 | 44.58 | 4.55 | 221.19 |
| ["9.22","14.84","19.84","24.84","29.84","34.84","39.84","44.84","49.84"] | 29.77 | 49.84 | 9.22 | 267.94 |
| ["8.75","13.36","18.36","23.36","28.36","33.36","38.36","43.36","48.36"] | 28.4 | 48.36 | 8.75 | 255.63 |
| ["8.47","14.76","19.76","24.76","29.76","34.76","39.76","44.76","49.76"] | 29.62 | 49.76 | 8.47 | 266.55 |
| ["4.93","9.99","14.99","19.99","24.99","29.99","34.99","39.99","44.99"] | 24.98 | 44.99 | 4.93 | 224.85 |
| ["4.65","14.98","19.98","24.98","29.98","34.98","39.98","44.98","49.98"] | 29.39 | 49.98 | 4.65 | 264.49 |
| ["7.80","14.62","19.62","24.62","29.62","34.62","39.62","44.62","49.62"] | 29.42 | 49.62 | 7.8 | 264.76 |
| ["9.32","9.96","14.96","19.96","24.96","29.96","34.96","39.96","44.96"] | 25.44 | 44.96 | 9.32 | 229 |

Results for the statistical functions:

| myArray2 | arrayMode | arrayStDv | arrayVar |
|---|---|---|---|
| | | | |

| | | |
|---|---|---|
| ["8.29","9.63","14.63","19.63","24.63","29.63","34.63","39.63","44.63"] | 12.32 | 151.72 |
| ["8.32","14.01","19.01","24.01","29.01","34.01","39.01","44.01","49.01"] | 13.03 | 169.78 |
| ["4.55","9.58","14.58","19.58","24.58","29.58","34.58","39.58","44.58"] | 12.92 | 166.8 |
| ["9.22","14.84","19.84","24.84","29.84","34.84","39.84","44.84","49.84"] | 13.02 | 169.46 |
| ["8.75","13.36","18.36","23.36","28.36","33.36","38.36","43.36","48.36"] | 12.84 | 164.95 |
| ["8.47","14.76","19.76","24.76","29.76","34.76","39.76","44.76","49.76"] | 13.14 | 172.56 |
| ["4.93","9.99","14.99","19.99","24.99","29.99","34.99","39.99","44.99"] | 12.92 | 166.93 |
| ["4.65","14.98","19.98","24.98","29.98","34.98","39.98","44.98","49.98"] | 13.9 | 193.16 |
| ["7.80","14.62","19.62","24.62","29.62","34.62","39.62","44.62","49.62"] | 13.23 | 175.08 |
| ["9.32","9.96","14.96","19.96","24.96","29.96","34.96","39.96","44.96"] | 12.21 | 149.17 |

Since all values are unique within an individual array, there is no most common value in any of them, which yields empty values for the `arrayMode` column.

# LISTVAR Function

Computes the variance of all numeric values found in input array. Input can be an array literal, a column of arrays, or a function returning an array. Input values must be of Integer or Decimal type.

When this function is invoked, all of the values in the input array are passed to the corresponding columnar function. Some restrictions may apply. See *VAR Function*.

**Wrangle vs. SQL:** This function is part of Wrangle , a proprietary data transformation language. Wrangle is not SQL. For more information, see *Wrangle Language*.

## Basic Usage

**Literal example:**

```
listvar([0,0,2,4,6,8,10,12,14,16,18,20])
```

**Output:** Returns the variance of all values in the literal array: `48.33333333`.

**Column example:**

```
listvar(myArray)
```

**Output:** Returns the variance of all values in the arrays of the `myArray` column.

## Syntax and Arguments

```
listvar(array_ref)
```

| Argument | Required? | Data Type | Description |
|----------|-----------|-----------|-------------|
| array_ref | Y | Array | Array literal, reference to column containing arrays, or function returning an array |

For more information on syntax standards, see *Language Documentation Syntax Notes*.

**array_ref**

Reference to an array can be an array literal, function returning an array, or a single column containing arrays.

- If the input is not a valid numeric array, null values are returned.
- Non-numerical values within an input array are not factored in the computation.
- Multiple columns and wildcards are not supported.

**Usage Notes:**

| Required? | Data Type | Example Value |
|-----------|-----------|---------------|
| Yes | Array | `myArray` |

## Examples

> **Tip:** For additional examples, see *Common Tasks*.

## Example - Math functions for lists (arrays)

This example describes how to generate random array (list) data and then to apply statistical functions specifically created for arrays.

### Functions:

| Item | Description |
|------|-------------|
| LISTSUM Function | Computes the sum of all numeric values found in input array. Input can be an array literal, a column of arrays, or a function returning an array. Input values must be of Integer or Decimal type. |
| LISTMIN Function | Computes the minimum of all numeric values found in input array. Input can be an array literal, a column of arrays, or a function returning an array. Input values must be of Integer or Decimal type. |
| LISTMAX Function | Computes the maximum of all numeric values found in input array. Input can be an array literal, a column of arrays, or a function returning an array. Input values must be of Integer or Decimal type. |
| LISTAVERAGE Function | Computes the average of all numeric values found in input array. Input can be an array literal, a column of arrays, or a function returning an array. Input values must be of Integer or Decimal type. |
| LISTVAR Function | Computes the variance of all numeric values found in input array. Input can be an array literal, a column of arrays, or a function returning an array. Input values must be of Integer or Decimal type. |
| LISTSTDEV Function | Computes the standard deviation of all numeric values found in input array. Input can be an array literal, a column of arrays, or a function returning an array. Input values must be of Integer or Decimal type. |
| LISTMODE Function | Computes the most common value of all numeric values found in input array. Input can be an array literal, a column of arrays, or a function returning an array. Input values must be of Integer or Decimal type. |

Also:

| Item | Description |
|------|-------------|
| RANGE Function | Computes an array of integers, from a beginning integer to an end (stop) integer, stepping by a third parameter. |
| RAND Function | The RAND function generates a random real number between 0 and 1. The function accepts an optional integer parameter, which causes the same set of random numbers to be generated with each job execution. |
| ROUND Function | Rounds input value to the nearest integer. Input can be an Integer, a Decimal, a column reference, or an expression. Optional second argument can be used to specify the number of digits to which to round. |

### Source:

For this example, you can generate some randomized data using the following steps. First, you need to seed an array with a range of values using the RANGE function:

| Transformation Name | New formula |
|---------------------|-------------|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | RANGE(5, 50, 5) |
| Parameter: New column name | 'myArray1' |

Then, unpack this array, so you can add a random factor:

| Transformation Name | Unnest Objects into columns |
|---------------------|------------------------------|
| Parameter: Column | myArray1 |
| Parameter: Paths to elements | '[0]', '[1]', '[2]', '[3]', '[4]', '[5]', '[6]', |

| | '[7]', '[8]', '[9]' |
|---|---|
| **Parameter: Remove elements from original** | `true` |
| **Parameter: Include original column name** | `true` |

Add the randomizing factor. Here, you are adding randomization around individual values: x-1 < x < x+4.

| | |
|---|---|
| **Transformation Name** | `Edit column with formula` |
| **Parameter: Columns** | `myArray1_0~myArray1_8` |
| **Parameter: Formula** | `IF(RAND() > 0.5, $col + (5 * RAND()), $col - RAND())` |

To make the numbers easier to manipulate, you can round them to two decimal places:

| | |
|---|---|
| **Transformation Name** | `Edit column with formula` |
| **Parameter: Columns** | `myArray1_0~myArray1_8` |
| **Parameter: Formula** | `ROUND($col, 2)` |

Renest these columns into an array:

| | |
|---|---|
| **Transformation Name** | `Nest columns into Objects` |
| **Parameter: Columns** | `myArray1_0, myArray1_1, myArray1_2, myArray1_3, myArray1_4, myArray1_5, myArray1_6, myArray1_7, myArray1_8` |
| **Parameter: Nest columns to** | `Array` |
| **Parameter: New column name** | `'myArray2'` |

Delete the unused columns:

| | |
|---|---|
| **Transformation Name** | `Delete columns` |
| **Parameter: Columns** | `myArray1_0~myArray1_8,myArray1` |
| **Parameter: Action** | `Delete selected columns` |

Your data should look similar to the following:

| myArray2 |
|---|
| ["8.29","9.63","14.63","19.63","24.63","29.63","34.63","39.63","44.63"] |
| ["8.32","14.01","19.01","24.01","29.01","34.01","39.01","44.01","49.01"] |
| ["4.55","9.58","14.58","19.58","24.58","29.58","34.58","39.58","44.58"] |
| ["9.22","14.84","19.84","24.84","29.84","34.84","39.84","44.84","49.84"] |
| ["8.75","13.36","18.36","23.36","28.36","33.36","38.36","43.36","48.36"] |
| ["8.47","14.76","19.76","24.76","29.76","34.76","39.76","44.76","49.76"] |
| ["4.93","9.99","14.99","19.99","24.99","29.99","34.99","39.99","44.99"] |

| ["4.65","14.98","19.98","24.98","29.98","34.98","39.98","44.98","49.98"] |
|---|
| ["7.80","14.62","19.62","24.62","29.62","34.62","44.62","49.62"] |
| ["9.32","9.96","14.96","19.96","24.96","29.96","34.96","39.96","44.96"] |

**Transformation:**

These steps demonstrate the individual math functions that you can apply to your list data without unnesting it:

> **NOTE:** The NUMFORMAT function has been wrapped around each list function to account for any floating-point errors or additional digits in the results.

Sum of all values in the array (list):

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | NUMFORMAT(LISTSUM(myArray2), '#.##') |
| Parameter: New column name | 'arraySum' |

Minimum of all values in the array (list):

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | NUMFORMAT(LISTMIN(myArray2), '#.##') |
| Parameter: New column name | 'arrayMin' |

Maximum of all values in the array (list):

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | NUMFORMAT(LISTMAX(myArray2), '#.##') |
| Parameter: New column name | 'arrayMax' |

Average of all values in the array (list):

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | NUMFORMAT(LISTAVERAGE(myArray2), '#.##') |
| Parameter: New column name | 'arrayAvg' |

Variance of all values in the array (list):

| Transformation Name | New formula |
| --- | --- |
| Parameter: Formula type | Single row formula |
| Parameter: Formula | NUMFORMAT(LISTVAR(myArray2), '#.##') |
| Parameter: New column name | 'arrayVar' |

Standard deviation of all values in the array (list):

| Transformation Name | New formula |
| --- | --- |
| Parameter: Formula type | Single row formula |
| Parameter: Formula | NUMFORMAT(LISTSTDEV(myArray2), '#.##') |
| Parameter: New column name | 'arrayStDv' |

Mode (most common value) of all values in the array (list):

| Transformation Name | New formula |
| --- | --- |
| Parameter: Formula type | Single row formula |
| Parameter: Formula | NUMFORMAT(LISTMODE(myArray2), '#.##') |
| Parameter: New column name | 'arrayMode' |

**Results:**

Results for the first four math functions:

| myArray2 | arrayAvg | arrayMax | arrayMin | arraySum |
| --- | --- | --- | --- | --- |
| ["8.29","9.63","14.63","19.63","24.63","29.63","34.63","39.63","44.63"] | 25.04 | 44.63 | 8.29 | 225.33 |
| ["8.32","14.01","19.01","24.01","29.01","34.01","39.01","44.01","49.01"] | 28.93 | 49.01 | 8.32 | 260.4 |
| ["4.55","9.58","14.58","19.58","24.58","29.58","34.58","39.58","44.58"] | 24.58 | 44.58 | 4.55 | 221.19 |
| ["9.22","14.84","19.84","24.84","29.84","34.84","39.84","44.84","49.84"] | 29.77 | 49.84 | 9.22 | 267.94 |
| ["8.75","13.36","18.36","23.36","28.36","33.36","38.36","43.36","48.36"] | 28.4 | 48.36 | 8.75 | 255.63 |
| ["8.47","14.76","19.76","24.76","29.76","34.76","39.76","44.76","49.76"] | 29.62 | 49.76 | 8.47 | 266.55 |
| ["4.93","9.99","14.99","19.99","24.99","29.99","34.99","39.99","44.99"] | 24.98 | 44.99 | 4.93 | 224.85 |
| ["4.65","14.98","19.98","24.98","29.98","34.98","39.98","44.98","49.98"] | 29.39 | 49.98 | 4.65 | 264.49 |
| ["7.80","14.62","19.62","24.62","29.62","34.62","39.62","44.62","49.62"] | 29.42 | 49.62 | 7.8 | 264.76 |
| ["9.32","9.96","14.96","19.96","24.96","29.96","34.96","39.96","44.96"] | 25.44 | 44.96 | 9.32 | 229 |

Results for the statistical functions:

| myArray2 | arrayMode | arrayStDv | arrayVar |
| --- | --- | --- | --- |

| | | |
|---|---|---|
| ["8.29","9.63","14.63","19.63","24.63","29.63","34.63","39.63","44.63"] | 12.32 | 151.72 |
| ["8.32","14.01","19.01","24.01","29.01","34.01","39.01","44.01","49.01"] | 13.03 | 169.78 |
| ["4.55","9.58","14.58","19.58","24.58","29.58","34.58","39.58","44.58"] | 12.92 | 166.8 |
| ["9.22","14.84","19.84","24.84","29.84","34.84","39.84","44.84","49.84"] | 13.02 | 169.46 |
| ["8.75","13.36","18.36","23.36","28.36","33.36","38.36","43.36","48.36"] | 12.84 | 164.95 |
| ["8.47","14.76","19.76","24.76","29.76","34.76","39.76","44.76","49.76"] | 13.14 | 172.56 |
| ["4.93","9.99","14.99","19.99","24.99","29.99","34.99","39.99","44.99"] | 12.92 | 166.93 |
| ["4.65","14.98","19.98","24.98","29.98","34.98","39.98","44.98","49.98"] | 13.9 | 193.16 |
| ["7.80","14.62","19.62","24.62","29.62","34.62","39.62","44.62","49.62"] | 13.23 | 175.08 |
| ["9.32","9.96","14.96","19.96","24.96","29.96","34.96","39.96","44.96"] | 12.21 | 149.17 |

Since all values are unique within an individual array, there is no most common value in any of them, which yields empty values for the `arrayMode` column.

# Type Functions

Use the following functions to identify if your data matches a specified data type or is missing or null.

> **Tip:** It's typically easier to review and manipulate data types and their valid, mismatched, or missing values through the application. At the top of each column in the data grid, you can review the data quality bar, which contains color-coded identifiers of valid, invalid, or missing data for the currently selected data type. You can click each of these bars to perform transformations on each category of data, or you can select a new data type from the data type drop-down for the column to review data quality for the column under a different data type.

When making references to a data type within your Wrangle transforms, you must reference the appropriate data type key. See *Valid Data Type Strings*.

# NULL Function

The `NULL` function generates null values.

- The `ISNULL` function tests for the presence of null values. See *ISNULL Function*.
- Null values are different from missing values.
    - To test for missing values, see *ISMISSING Function*.

**Wrangle vs. SQL:** This function is part of Wrangle , a proprietary data transformation language. Wrangle is not SQL. For more information, see *Wrangle Language*.

## Basic Usage

```
null()
```

**Output:** Returns a null value.

```
if((isnull(FirstName) || isnull(LastName)), null(), &apos;not null&apos;) as:&apos;
status&apos;
```

**Output:** If there are null values in either the `FirstName` or `LastName` column, generate a null value in the `status` column. Otherwise, the returned value is `not null`.

## Syntax and Arguments

There are no arguments for this function.

## Examples

> **Tip:** For additional examples, see *Common Tasks*.

**Example - Type check functions**

This example illustrates how various type checking functions can be applied to your data.

**Functions**:

| Item | Description |
|------|-------------|
| VALID Function | Tests whether a set of values is valid for a specified data type and is not a null value. |
| ISMISMATCHED Function | Tests whether a set of values is not valid for a specified data type. |
| ISMISSING Function | The `ISMISSING` function tests whether a column of values is missing or null. For input column references, this function returns `true` or `false`. |
| ISNULL Function | The `ISNULL` function tests whether a column of values contains null values. For input column references, this function returns `true` or `false`. |
| NULL Function | The `NULL` function generates null values. |

**Source:**

Some source values that should match the State and Integer data types:

| State | Qty |
|-------|-----|
| CA | 10 |
| OR | -10 |
| WA | 2.5 |
| ZZ | 15 |
| ID | |
| | 4 |

**Transformation:**

**Invalid State values**: You can test for invalid values for State using the following:

| Transformation Name | New formula |
|---------------------|-------------|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | ISMISMATCHED (State, 'State') |

The above transform flags rows 4 and 6 as mismatched.

> **NOTE:** A missing value is not valid for a type, including String type.

**Invalid Integer values:** You can test for valid matches for Qty using the following:

| Transformation Name | New formula |
|---------------------|-------------|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | (ISVALID (Qty, 'Integer') && (Qty > 0)) |
| Parameter: New column name | 'valid_Qty' |

The above transform flags as valid all rows where the `Qty` column is a valid integer that is greater than zero.

**Missing values:** The following transform tests for the presence of missing values in either column:

| Transformation Name | New formula |
|---------------------|-------------|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | (ISMISSING(State) \|\| ISMISSING(Qty)) |
| Parameter: New column name | 'missing_State_Qty' |

After re-organizing the columns using the `move` transform, the dataset should now look like the following:

| State | Qty | mismatched_State | valid_Qty | missing_State_Qty |
|-------|-----|------------------|-----------|-------------------|
| CA | 10 | false | true | false |
| OR | -10 | false | false | false |
| WA | 2.5 | false | false | false |

| | | | | |
|---|---|---|---|---|
| ZZ | 15 | true | true | false |
| ID | | false | false | true |
| | 4 | false | true | true |

Since the data does not contain null values, the following transform generates null values based on the preceding criteria:

| | |
|---|---|
| **Transformation Name** | `New formula` |
| **Parameter: Formula type** | `Single row formula` |
| **Parameter: Formula** | `((mismatched_State == 'true') \|\| (valid_Qty == 'false') \|\| (missing_State_Qty == 'true')) ? NULL() : 'ok'` |
| **Parameter: New column name** | `'status'` |

You can then use the `ISNULL` check to remove the rows that fail the above test:

| | |
|---|---|
| **Transformation Name** | `Filter rows` |
| **Parameter: Condition** | `Custom formula` |
| **Parameter: Type of formula** | `Custom single` |
| **Parameter: Condition** | `ISNULL('status')` |
| **Parameter: Action** | `Delete matching rows` |

**Results:**

Based on the above tests, the output dataset contains one row:

| State | Qty | mismatched_State | valid_Qty | missing_State_Qty | status |
|---|---|---|---|---|---|
| CA | 10 | false | true | false | ok |

# IFNULL Function

**Contents:**

---

The `IFNULL` function writes out a specified value if the source value is a null. Otherwise, it writes the source value. Input can be a literal, a column reference, or a function.

- The `NULL` function generates null values. See *NULL Function*.
- The `ISNULL` function simply tests if a value is null. See *ISNULL Function*.
- Null values are different from missing values.
    - To test for missing values, see `IFMISSING` Function. See *IFMISSING Function*.

**Wrangle vs. SQL:** This function is part of Wrangle , a proprietary data transformation language. Wrangle is not SQL. For more information, see *Wrangle Language*.

## Basic Usage

```
ifnull(my_score,&apos;0&apos;)
```

**Output:** Returns the value `0` if the value in `my_score` is a null.

## Syntax and Arguments

```
ifnull(column_string, computed_value)
```

| Argument | Required? | Data Type | Description |
|----------|-----------|-----------|-------------|
| source_value | Y | string | Name of column, string literal or function to be tested |
| output_value | y | string | String literal value to write |

For more information on syntax standards, see *Language Documentation Syntax Notes*.

### source_value

Name of the column, string literal, or function to be tested for null values.

- Missing literals or column values generate missing string results.
- Multiple columns and wildcards are not supported.

**Usage Notes:**

| Required? | Data Type | Example Value |
|-----------|-----------|---------------|
| | | |

| Yes | String literal, column reference, or function | `myColumn` |

### output_value

The output value to write if the tested value returns a null value.

**Usage Notes:**

| Required? | Data Type | Example Value |
|-----------|-----------|---------------|
| Yes | String or numeric literal | `'Null input'` |

## Examples

> **Tip:** For additional examples, see *Common Tasks*.

### Example - IF* functions for data type validation

This example illustrates how to use the IF* functions for data type validation.

**Functions**:

| Item | Description |
|------|-------------|
| IFNULL Function | The `IFNULL` function writes out a specified value if the source value is a null. Otherwise, it writes the source value. Input can be a literal, a column reference, or a function. |
| IFMISSING Function | The `IFMISSING` function writes out a specified value if the source value is a null or missing value. Otherwise, it writes the source value. Input can be a literal, a column reference, or a function. |
| IFMISMATCHED Function | The `IFMISMATCHED` function writes out a specified value if the input expression does not match the specified data type or typing array. Otherwise, it writes the source value. Input can be a literal, a column reference, or a function. |
| IFVALID Function | The `IFVALID` function writes out a specified value if the input expression matches the specified data type. Otherwise, it writes the source value. Input can be a literal, a column reference, or a function. |
| MERGE Function | Merges two or more columns of String type to generate output of String type. Optionally, you can insert a delimiter between the merged values. |

**Source:**

The following simple table lists zip codes by customer identifier:

| custId | custZip |
|--------|---------|
| C001 | 98123 |
| C002 | 94105 |
| C003 | 12415 |
| C004 | 12451-2234 |
| C005 | 12441-298 |
| C006 | |
| C007 | |
| C008 | 1242 |

| C009 | 1104 |
|------|------|

**Transformation:**

When the above is imported into the Transformer page, you notice the following:

- The custZip column is typed as Integer.
- There are two missing and two mismatched values in the custZip column.

First, you test for valid values in the custZip column. Using the IFVALID function, you can validate against any data type:

| | |
|---|---|
| **Transformation Name** | New formula |
| **Parameter: Formula type** | Single row formula |
| **Parameter: Formula** | IFVALID(custZip, 'Zipcode', 'ok') |
| **Parameter: New column name** | 'status' |

**Fix four-digit zips:** In the status column are instances of ok for the top four rows. You notice that the bottom two rows contain four-digit codes.

Since the custZip values were originally imported as Integer, any leading 0 values are deleted. In this case, you can add back the leading zero. Before the previous step, change the data type of zip to String and insert the following:

| | |
|---|---|
| **Transformation Name** | New formula |
| **Parameter: Formula type** | Single row formula |
| **Parameter: Formula** | IF(LEN(custZip)==4,'0','') |
| **Parameter: New column name** | 'FourDigitZip' |

| | |
|---|---|
| **Transformation Name** | New formula |
| **Parameter: Formula type** | Single row formula |
| **Parameter: Formula** | MERGE([FourDigitZip,custZip]) |
| **Parameter: New column name** | 'custZip2' |

| | |
|---|---|
| **Transformation Name** | Edit column with formula |
| **Parameter: Columns** | zip |
| **Parameter: Formula** | custZip2 |

| | |
|---|---|
| **Transformation Name** | Delete columns |
| **Parameter: Columns** | FourDigitZip,custZip2 |
| **Parameter: Action** | Delete selected columns |

Now, when you click the last recipe step, you should see that two more rows in `status` are listed as `Ok`.

For the zip code with the three-digit extension, you can simply remove that extension to make it valid. Click the step above the last one. In the data grid, highlight the value. Click the Replace suggestion card. Select the option that uses the following for the matching pattern:

```
'-{digit}{3}{end}'
```

The above means that all three-digit extensions are deleted from the zip. You can do the same for any two- and one-digit extensions, although there are none in this sample.

**Missing and null values:** Now, you need to address how to handle missing and null values. The `IFMISSING` tests for both missing and null values, while the `IFNULL` tests just for null values. In this example, you want to delete null values, which could mean that the data for that row is malformed and to write a status of `missing` for missing values.

Click above the last line in the recipe to insert the following:

| | |
|---|---|
| **Transformation Name** | Edit column with formula |
| **Parameter: Columns** | custZip |
| **Parameter: Formula** | IFNULL(custZip, 'xxxxx') |

| | |
|---|---|
| **Transformation Name** | Edit column with formula |
| **Parameter: Columns** | custZip |
| **Parameter: Formula** | IFMISSING(custZip, '00000') |

Now, when you click the last line of the recipe, only the null value is listed as having a status other than `ok`. You can use the following to remove this row and all like it:

| | |
|---|---|
| **Transformation Name** | Filter rows |
| **Parameter: Condition** | Custom formula |
| **Parameter: Type of formula** | Custom single |
| **Parameter: Condition** | (status == 'xxxxx') |
| **Parameter: Action** | Delete matching rows |

**Results:**

| custId | custZip | status |
|---|---|---|
| C001 | 98123 | ok |
| C002 | 94105 | ok |
| C003 | 12415 | ok |
| C004 | 12451-2234 | ok |
| C005 | 12441-298 | ok |
| C006 | 00000 | ok |
| C008 | 1242 | ok |

| | | |
|------|------|----|
| C009 | 1104 | ok |

As an exercise, you might repeat the above steps starting with the `IFMISMATCHED` function determining the value in the `status` column:

| | |
|---|---|
| **Transformation Name** | `New formula` |
| **Parameter: Formula type** | `Single row formula` |
| **Parameter: Formula** | `IFMISMATCHED(custZip, 'Zipcode', 'mismatched')` |
| **Parameter: New column name** | `'status'` |

# IFMISSING Function

**Contents:**

The `IFMISSING` function writes out a specified value if the source value is a null or missing value. Otherwise, it writes the source value. Input can be a literal, a column reference, or a function.

- The `ISMISSING` function simply tests if a value is missing. See *ISMISSING Function*.
- Missing values are different from null values. To test for null values, see *IFNULL Function*.

> **Tip:** Since this function captures both missing and null values, you may first wish to address the rows with null values using the `IFNULL` or `ISNULL` functions. Any remaining rows that are matched based on this function are exclusively missing values.

**Wrangle vs. SQL:** This function is part of Wrangle , a proprietary data transformation language. Wrangle is not SQL. For more information, see *Wrangle Language*.

## Basic Usage

```
ifmissing(my_score,&apos;0&apos;)
```

**Output:** Generates a new column called, `final_score`, which contains the value `0` if the value in `my_score` is a null or missing value.

## Syntax and Arguments

```
ifmissing(column_string, computed_value)
```

| Argument | Required? | Data Type | Description |
|----------|-----------|-----------|-------------|
| source_value | Y | string | Name of column, string literal or function to be tested |
| output_value | y | string | String literal value to write |

For more information on syntax standards, see *Language Documentation Syntax Notes*.

**source_value**

Name of the column, string literal, or function to be tested for missing values.

- Missing literals or column values generate missing string results.
- Multiple columns and wildcards are not supported.

**Usage Notes:**

| Required? | Data Type | Example Value |
|---|---|---|
| Yes | String literal, column reference, or function | `myColumn` |

**output_value**

The output value to write if the tested value returns a null or missing value.

**Usage Notes:**

| Required? | Data Type | Example Value |
|---|---|---|
| Yes | String or numeric literal | `'Missing input'` |

## Examples

> **Tip:** For additional examples, see *Common Tasks*.

**Example - IF\* functions for data type validation**

This example illustrates how to use the IF\* functions for data type validation.

**Functions**:

| Item | Description |
|---|---|
| IFNULL Function | The `IFNULL` function writes out a specified value if the source value is a null. Otherwise, it writes the source value. Input can be a literal, a column reference, or a function. |
| IFMISSING Function | The `IFMISSING` function writes out a specified value if the source value is a null or missing value. Otherwise, it writes the source value. Input can be a literal, a column reference, or a function. |
| IFMISMATCHED Function | The `IFMISMATCHED` function writes out a specified value if the input expression does not match the specified data type or typing array. Otherwise, it writes the source value. Input can be a literal, a column reference, or a function. |
| IFVALID Function | The `IFVALID` function writes out a specified value if the input expression matches the specified data type. Otherwise, it writes the source value. Input can be a literal, a column reference, or a function. |
| MERGE Function | Merges two or more columns of String type to generate output of String type. Optionally, you can insert a delimiter between the merged values. |

**Source:**

The following simple table lists zip codes by customer identifier:

| custId | custZip |
|---|---|
| C001 | 98123 |
| C002 | 94105 |
| C003 | 12415 |
| C004 | 12451-2234 |
| C005 | 12441-298 |
| C006 | |

| C007 |  |
|------|------|
| C008 | 1242 |
| C009 | 1104 |

**Transformation:**

When the above is imported into the Transformer page, you notice the following:

- The custZip column is typed as Integer.
- There are two missing and two mismatched values in the custZip column.

First, you test for valid values in the custZip column. Using the IFVALID function, you can validate against any data type:

| Transformation Name | New formula |
|---------------------|-------------|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | IFVALID(custZip, 'Zipcode', 'ok') |
| Parameter: New column name | 'status' |

**Fix four-digit zips:** In the status column are instances of ok for the top four rows. You notice that the bottom two rows contain four-digit codes.

Since the custZip values were originally imported as Integer, any leading 0 values are deleted. In this case, you can add back the leading zero. Before the previous step, change the data type of zip to String and insert the following:

| Transformation Name | New formula |
|---------------------|-------------|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | IF(LEN(custZip)==4,'0','') |
| Parameter: New column name | 'FourDigitZip' |

| Transformation Name | New formula |
|---------------------|-------------|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | MERGE([FourDigitZip,custZip]) |
| Parameter: New column name | 'custZip2' |

| Transformation Name | Edit column with formula |
|---------------------|--------------------------|
| Parameter: Columns | zip |
| Parameter: Formula | custZip2 |

| Transformation Name | Delete columns |
|---------------------|----------------|
|  |  |

| Parameter: Columns | FourDigitZip,custZip2 |
|---|---|
| Parameter: Action | Delete selected columns |

Now, when you click the last recipe step, you should see that two more rows in `status` are listed as `Ok`.

For the zip code with the three-digit extension, you can simply remove that extension to make it valid. Click the step above the last one. In the data grid, highlight the value. Click the Replace suggestion card. Select the option that uses the following for the matching pattern:

```
'-{digit}{3}{end}'
```

The above means that all three-digit extensions are deleted from the zip. You can do the same for any two- and one-digit extensions, although there are none in this sample.

**Missing and null values:** Now, you need to address how to handle missing and null values. The `IFMISSING` tests for both missing and null values, while the `IFNULL` tests just for null values. In this example, you want to delete null values, which could mean that the data for that row is malformed and to write a status of `missing` for missing values.

Click above the last line in the recipe to insert the following:

| Transformation Name | Edit column with formula |
|---|---|
| Parameter: Columns | custZip |
| Parameter: Formula | IFNULL(custZip, 'xxxxx') |

| Transformation Name | Edit column with formula |
|---|---|
| Parameter: Columns | custZip |
| Parameter: Formula | IFMISSING(custZip, '00000') |

Now, when you click the last line of the recipe, only the null value is listed as having a status other than `ok`. You can use the following to remove this row and all like it:

| Transformation Name | Filter rows |
|---|---|
| Parameter: Condition | Custom formula |
| Parameter: Type of formula | Custom single |
| Parameter: Condition | (status == 'xxxxx') |
| Parameter: Action | Delete matching rows |

**Results:**

| custId | custZip | status |
|---|---|---|
| C001 | 98123 | ok |
| C002 | 94105 | ok |
| C003 | 12415 | ok |
| C004 | 12451-2234 | ok |

| | | |
|---|---|---|
| C005 | 12441-298 | ok |
| C006 | 00000 | ok |
| C008 | 1242 | ok |
| C009 | 1104 | ok |

As an exercise, you might repeat the above steps starting with the `IFMISMATCHED` function determining the value in the `status` column:

| | |
|---|---|
| **Transformation Name** | `New formula` |
| **Parameter: Formula type** | `Single row formula` |
| **Parameter: Formula** | `IFMISMATCHED(custZip, 'Zipcode', 'mismatched')` |
| **Parameter: New column name** | `'status'` |

# IFMISMATCHED Function

**Contents:**

---

The `IFMISMATCHED` function writes out a specified value if the input expression does not match the specified data type or typing array. Otherwise, it writes the source value. Input can be a literal, a column reference, or a function.

The `ISMISMATCHED` function simply tests if a value is mismatched. See *ISMISMATCHED Function*.

**Wrangle vs. SQL:** This function is part of Wrangle , a proprietary data transformation language. Wrangle is not SQL. For more information, see *Wrangle Language*.

## Basic Usage

**Basic data type:**

```
ifmismatched(my_ssn,'SSN', 'XXX-XX-XXXX')
```

**Output:** Returns the value `XXX-XX-XXXX` if the value in `my_ssn` does not match the `SSN` data type.

**Data type with formatting options:**

For data types with formatting options, such as Datetime, you can specify the format using an array, as in the following:

```
ifmismatched(month_Date, ['Datetime','mm-dd-yy','mm*dd*yy'],
null())
```

**Output:** Returns null if values in `month_Date` are mismatched against Datetime values in the `mm-dd-yy` or `mm*dd*yy` formats.

## Syntax and Arguments

```
ifmismatched(column_string, data_type_literal, computed_value)
```

| Argument | Required? | Data Type | Description |
|----------|-----------|-----------|-------------|
| source_value | Y | string | Name of column, string literal or function to be tested |
| datatype_literal | Y | string | String literal or array that identifies the data type against which to validate the source values |
| output_value | y | string | String literal value to write |

For more information on syntax standards, see *Language Documentation Syntax Notes*.

**source_value**

Name of the column, string literal, or function to be tested for data type matches.

- Missing literals or column values generate missing string results.
- Multiple columns and wildcards are not supported.

**Usage Notes:**

| Required? | Data Type | Example Value |
|---|---|---|
| Yes | String literal, column reference, or function | `myColumn` |

**datatype_literal**

Literal value or string array that identifies data type to which to validate the source column or string.

- Column references are not supported.

**Usage Notes:**

| Required? | Data Type | Example Value |
|---|---|---|
| Yes | String literal | `'Integer'` |

**Valid data type strings:**

When referencing a data type within a transform, you can use the following strings to identify each type:

> **NOTE:** In Wrangle transforms, these values are case-sensitive.

> **NOTE:** When specifying a data type by name, you must use the String value listed below. The Data Type value is the display name for the type.

| Data Type | String |
|---|---|
| String | `'String'` |
| Integer | `'Integer'` |
| Decimal | `'Float'` |
| Boolean | `'Bool'` |
| Social Security Number | `'SSN'` |
| Phone Number | `'Phone'` |
| Email Address | `'Emailaddress'` |
| Credit Card | `'Creditcard'` |
| | |

| Gender | `'Gender'` |
|---|---|
| Object | `'Map'` |
| Array | `'Array'` |
| IP Address | `'Ipaddress'` |
| URL | `'Url'` |
| HTTP Code | `'Httpcodes'` |
| Zip Code | `'Zipcode'` |
| State | `'State'` |
| Date / Time | `'Datetime'` |

**output_value**

The output value to write if the tested value is mismatched for the specified data type.

**Usage Notes:**

| Required? | Data Type | Example Value |
|---|---|---|
| Yes | String or numeric literal | `'Data type mismatch'` |

## Examples

> **Tip:** For additional examples, see *Common Tasks*.

### Example - IF* functions for data type validation

This example illustrates how to use the IF* functions for data type validation.

**Functions**:

| Item | Description |
|---|---|
| IFNULL Function | The `IFNULL` function writes out a specified value if the source value is a null. Otherwise, it writes the source value. Input can be a literal, a column reference, or a function. |
| IFMISSING Function | The `IFMISSING` function writes out a specified value if the source value is a null or missing value. Otherwise, it writes the source value. Input can be a literal, a column reference, or a function. |
| IFMISMATC HED Function | The `IFMISMATCHED` function writes out a specified value if the input expression does not match the specified data type or typing array. Otherwise, it writes the source value. Input can be a literal, a column reference, or a function. |
| IFVALID Function | The `IFVALID` function writes out a specified value if the input expression matches the specified data type. Otherwise, it writes the source value. Input can be a literal, a column reference, or a function. |
| MERGE Function | Merges two or more columns of String type to generate output of String type. Optionally, you can insert a delimiter between the merged values. |

**Source:**

The following simple table lists zip codes by customer identifier:

| custId | custZip |
|---|---|

| | |
|------|------------|
| C001 | 98123 |
| C002 | 94105 |
| C003 | 12415 |
| C004 | 12451-2234 |
| C005 | 12441-298 |
| C006 | |
| C007 | |
| C008 | 1242 |
| C009 | 1104 |

**Transformation:**

When the above is imported into the Transformer page, you notice the following:

- The `custZip` column is typed as Integer.
- There are two missing and two mismatched values in the `custZip` column.

First, you test for valid values in the `custZip` column. Using the `IFVALID` function, you can validate against any data type:

| | |
|------|------|
| **Transformation Name** | New formula |
| **Parameter: Formula type** | Single row formula |
| **Parameter: Formula** | IFVALID(custZip, 'Zipcode', 'ok') |
| **Parameter: New column name** | 'status' |

**Fix four-digit zips:** In the `status` column are instances of `ok` for the top four rows. You notice that the bottom two rows contain four-digit codes.

Since the `custZip` values were originally imported as Integer, any leading `0` values are deleted. In this case, you can add back the leading zero. Before the previous step, change the data type of `zip` to String and insert the following:

| | |
|------|------|
| **Transformation Name** | New formula |
| **Parameter: Formula type** | Single row formula |
| **Parameter: Formula** | IF(LEN(custZip)==4,'0','') |
| **Parameter: New column name** | 'FourDigitZip' |

| | |
|------|------|
| **Transformation Name** | New formula |
| **Parameter: Formula type** | Single row formula |
| **Parameter: Formula** | MERGE([FourDigitZip,custZip]) |
| **Parameter: New column name** | 'custZip2' |

| Transformation Name | Edit column with formula |
|---|---|
| Parameter: Columns | zip |
| Parameter: Formula | custZip2 |

| Transformation Name | Delete columns |
|---|---|
| Parameter: Columns | FourDigitZip,custZip2 |
| Parameter: Action | Delete selected columns |

Now, when you click the last recipe step, you should see that two more rows in `status` are listed as `Ok`.

For the zip code with the three-digit extension, you can simply remove that extension to make it valid. Click the step above the last one. In the data grid, highlight the value. Click the Replace suggestion card. Select the option that uses the following for the matching pattern:

```
'-{digit}{3}{end}'
```

The above means that all three-digit extensions are deleted from the zip. You can do the same for any two- and one-digit extensions, although there are none in this sample.

**Missing and null values:** Now, you need to address how to handle missing and null values.  The `IFMISSING` tests for both missing and null values, while the `IFNULL` tests just for null values. In this example, you want to delete null values, which could mean that the data for that row is malformed and to write a status of `missing` for missing values.

Click above the last line in the recipe to insert the following:

| Transformation Name | Edit column with formula |
|---|---|
| Parameter: Columns | custZip |
| Parameter: Formula | IFNULL(custZip, 'xxxxx') |

| Transformation Name | Edit column with formula |
|---|---|
| Parameter: Columns | custZip |
| Parameter: Formula | IFMISSING(custZip, '00000') |

Now, when you click the last line of the recipe, only the null value is listed as having a status other than `ok`. You can use the following to remove this row and all like it:

| Transformation Name | Filter rows |
|---|---|
| Parameter: Condition | Custom formula |
| Parameter: Type of formula | Custom single |
| Parameter: Condition | (status == 'xxxxx') |
| Parameter: Action | Delete matching rows |

**Results:**

| custId | custZip | status |
|--------|---------|--------|
| C001 | 98123 | ok |
| C002 | 94105 | ok |
| C003 | 12415 | ok |
| C004 | 12451-2234 | ok |
| C005 | 12441-298 | ok |
| C006 | 00000 | ok |
| C008 | 1242 | ok |
| C009 | 1104 | ok |

As an exercise, you might repeat the above steps starting with the `IFMISMATCHED` function determining the value in the `status` column:

| | |
|---|---|
| **Transformation Name** | `New formula` |
| **Parameter: Formula type** | `Single row formula` |
| **Parameter: Formula** | `IFMISMATCHED(custZip, 'Zipcode', 'mismatched')` |
| **Parameter: New column name** | `'status'` |

# IFVALID Function

**Contents:**

The `IFVALID` function writes out a specified value if the input expression matches the specified data type. Otherwise, it writes the source value. Input can be a literal, a column reference, or a function.

The `VALID` function simply tests if a value is valid. See *VALID Function*.

**Wrangle vs. SQL:** This function is part of Wrangle , a proprietary data transformation language. Wrangle is not SQL. For more information, see *Wrangle Language*.

## Basic Usage

```
ifvalid(myZip,'ZipCode', 'ok')
```

**Output:** Returns the value `ok` if the value in `myZip` matches the `ZipCode` data type.

**Data type with formatting options:**

For data types with formatting options, such as Datetime, you can specify the format using an array, as in the following:

```
ifvalid(myDate, ['Datetime','mm-dd-yy','mm*dd*yy'], 'true')
```

**Output:** Returns the value `true`, if the value in the `myDate` column is a valid Datetime value in `yy-mm-dd` or `yy*mm*dd` format.

## Syntax and Arguments

```
ifvalid(column_string, data_type_literal, computed_value)
```

| Argument | Required? | Data Type | Description |
|---|---|---|---|
| source_value | Y | string | Name of column, string literal or function to be tested |
| datatype_literal | Y | string | String literal that identifies the data type against which to validate the source values |
| output_value | y | string | String literal value to write |

For more information on syntax standards, see *Language Documentation Syntax Notes*.

**source_value**

Name of the column, string literal, or function to be tested for data type matches.

- Missing literals or column values generate missing string results.
- Multiple columns and wildcards are not supported.

**Usage Notes:**

| Required? | Data Type | Example Value |
|---|---|---|
| Yes | String literal, column reference, or function | `myColumn` |

**datatype_literal**

Literal value for data type to which to validate the source column or string.

- Column references are not supported.

**Usage Notes:**

| Required? | Data Type | Example Value |
|---|---|---|
| Yes | String literal | `'Integer'` |

**Valid data type strings:**

When referencing a data type within a transform, you can use the following strings to identify each type:

> **NOTE:** In Wrangle transforms, these values are case-sensitive.

> **NOTE:** When specifying a data type by name, you must use the String value listed below. The Data Type value is the display name for the type.

| Data Type | String |
|---|---|
| String | `'String'` |
| Integer | `'Integer'` |
| Decimal | `'Float'` |
| Boolean | `'Bool'` |
| Social Security Number | `'SSN'` |
| Phone Number | `'Phone'` |
| Email Address | `'Emailaddress'` |
| Credit Card | `'Creditcard'` |
| Gender | `'Gender'` |
| Object | `'Map'` |

| | |
|---|---|
| Array | `'Array'` |
| IP Address | `'Ipaddress'` |
| URL | `'Url'` |
| HTTP Code | `'Httpcodes'` |
| Zip Code | `'Zipcode'` |
| State | `'State'` |
| Date / Time | `'Datetime'` |

**output_value**

The output value to write if the tested value is valid for the specified data type.

**Usage Notes:**

| Required? | Data Type | Example Value |
|---|---|---|
| Yes | String or numeric literal | `'Data type mismatch'` |

## Examples

> **Tip:** For additional examples, see *Common Tasks*.

**Example - IF\* functions for data type validation**

This example illustrates how to use the IF\* functions for data type validation.

**Functions**:

| Item | Description |
|---|---|
| IFNULL Function | The `IFNULL` function writes out a specified value if the source value is a null. Otherwise, it writes the source value. Input can be a literal, a column reference, or a function. |
| IFMISSING Function | The `IFMISSING` function writes out a specified value if the source value is a null or missing value. Otherwise, it writes the source value. Input can be a literal, a column reference, or a function. |
| IFMISMATCHED Function | The `IFMISMATCHED` function writes out a specified value if the input expression does not match the specified data type or typing array. Otherwise, it writes the source value. Input can be a literal, a column reference, or a function. |
| IFVALID Function | The `IFVALID` function writes out a specified value if the input expression matches the specified data type. Otherwise, it writes the source value. Input can be a literal, a column reference, or a function. |
| MERGE Function | Merges two or more columns of String type to generate output of String type. Optionally, you can insert a delimiter between the merged values. |

**Source:**

The following simple table lists zip codes by customer identifier:

| custId | custZip |
|---|---|
| C001 | 98123 |
| C002 | 94105 |

| | |
|---|---|
| C003 | 12415 |
| C004 | 12451-2234 |
| C005 | 12441-298 |
| C006 | |
| C007 | |
| C008 | 1242 |
| C009 | 1104 |

**Transformation:**

When the above is imported into the Transformer page, you notice the following:

- The custZip column is typed as Integer.
- There are two missing and two mismatched values in the custZip column.

First, you test for valid values in the custZip column. Using the IFVALID function, you can validate against any data type:

| | |
|---|---|
| **Transformation Name** | New formula |
| **Parameter: Formula type** | Single row formula |
| **Parameter: Formula** | IFVALID(custZip, 'Zipcode', 'ok') |
| **Parameter: New column name** | 'status' |

**Fix four-digit zips:** In the status column are instances of ok for the top four rows. You notice that the bottom two rows contain four-digit codes.

Since the custZip values were originally imported as Integer, any leading 0 values are deleted. In this case, you can add back the leading zero. Before the previous step, change the data type of zip to String and insert the following:

| | |
|---|---|
| **Transformation Name** | New formula |
| **Parameter: Formula type** | Single row formula |
| **Parameter: Formula** | IF(LEN(custZip)==4,'0','') |
| **Parameter: New column name** | 'FourDigitZip' |

| | |
|---|---|
| **Transformation Name** | New formula |
| **Parameter: Formula type** | Single row formula |
| **Parameter: Formula** | MERGE([FourDigitZip,custZip]) |
| **Parameter: New column name** | 'custZip2' |

| | |
|---|---|
| **Transformation Name** | Edit column with formula |
| **Parameter: Columns** | zip |

| Parameter: Formula | `custZip2` |
|---|---|

| Transformation Name | `Delete columns` |
|---|---|
| Parameter: Columns | `FourDigitZip,custZip2` |
| Parameter: Action | `Delete selected columns` |

Now, when you click the last recipe step, you should see that two more rows in `status` are listed as `Ok`.

For the zip code with the three-digit extension, you can simply remove that extension to make it valid. Click the step above the last one. In the data grid, highlight the value. Click the Replace suggestion card. Select the option that uses the following for the matching pattern:

```
'-{digit}{3}{end}'
```

The above means that all three-digit extensions are deleted from the zip. You can do the same for any two- and one-digit extensions, although there are none in this sample.

**Missing and null values:** Now, you need to address how to handle missing and null values. The `IFMISSING` tests for both missing and null values, while the `IFNULL` tests just for null values. In this example, you want to delete null values, which could mean that the data for that row is malformed and to write a status of `missing` for missing values.

Click above the last line in the recipe to insert the following:

| Transformation Name | `Edit column with formula` |
|---|---|
| Parameter: Columns | `custZip` |
| Parameter: Formula | `IFNULL(custZip, 'xxxxx')` |

| Transformation Name | `Edit column with formula` |
|---|---|
| Parameter: Columns | `custZip` |
| Parameter: Formula | `IFMISSING(custZip, '00000')` |

Now, when you click the last line of the recipe, only the null value is listed as having a status other than `ok`. You can use the following to remove this row and all like it:

| Transformation Name | `Filter rows` |
|---|---|
| Parameter: Condition | `Custom formula` |
| Parameter: Type of formula | `Custom single` |
| Parameter: Condition | `(status == 'xxxxx')` |
| Parameter: Action | `Delete matching rows` |

**Results:**

| custId | custZip | status |
|---|---|---|
| C001 | 98123 | ok |

| C002 | 94105 | ok |
| C003 | 12415 | ok |
| C004 | 12451-2234 | ok |
| C005 | 12441-298 | ok |
| C006 | 00000 | ok |
| C008 | 1242 | ok |
| C009 | 1104 | ok |

As an exercise, you might repeat the above steps starting with the `IFMISMATCHED` function determining the value in the `status` column:

| Transformation Name | `New formula` |
|---|---|
| **Parameter: Formula type** | `Single row formula` |
| **Parameter: Formula** | `IFMISMATCHED(custZip, 'Zipcode', 'mismatched')` |
| **Parameter: New column name** | `'status'` |

# ISNULL Function

The `ISNULL` function tests whether a column of values contains null values. For input column references, this function returns `true` or `false`.

- The `NULL` function generates null values. See *NULL Function*.
- Null values are different from missing values.
    - To test for missing values, see *ISMISSING Function*.

**Wrangle vs. SQL:** This function is part of Wrangle , a proprietary data transformation language. Wrangle is not SQL. For more information, see *Wrangle Language*.

## Basic Usage

```
isnull(Qty)
```

**Output:** Returns `true` if the value in the `Qty` column is null.

## Syntax and Arguments

```
isnull(column_string)
```

| Argument | Required? | Data Type | Description |
|----------|-----------|-----------|-------------|
| column_string | Y | string | Name of column or string literal to be applied to the function |

For more information on syntax standards, see *Language Documentation Syntax Notes*.

**column_string**

Name of the column or string literal to be tested for null values.

- Missing literals or column values generate missing string results.
- Multiple columns and wildcards are not supported.

**Usage Notes:**

| Required? | Data Type | Example Value |
|-----------|-----------|---------------|
| Yes | String literal or column reference | `myColumn` |

**Valid data type strings:**

When referencing a data type within a transform, you can use the following strings to identify each type:

> **NOTE:** In Wrangle transforms, these values are case-sensitive.

> **NOTE:** When specifying a data type by name, you must use the String value listed below. The Data Type value is the display name for the type.

| Data Type | String |
|---|---|
| String | `'String'` |
| Integer | `'Integer'` |
| Decimal | `'Float'` |
| Boolean | `'Bool'` |
| Social Security Number | `'SSN'` |
| Phone Number | `'Phone'` |
| Email Address | `'Emailaddress'` |
| Credit Card | `'Creditcard'` |
| Gender | `'Gender'` |
| Object | `'Map'` |
| Array | `'Array'` |
| IP Address | `'Ipaddress'` |
| URL | `'Url'` |
| HTTP Code | `'Httpcodes'` |
| Zip Code | `'Zipcode'` |
| State | `'State'` |
| Date / Time | `'Datetime'` |

## Examples

> **Tip:** For additional examples, see *Common Tasks*.

### Example - Type check functions

This example illustrates how various type checking functions can be applied to your data.

**Functions**:

| Item | Description |
|---|---|
| VALID Function | Tests whether a set of values is valid for a specified data type and is not a null value. |
| ISMISMATCHED Function | Tests whether a set of values is not valid for a specified data type. |
| ISMISSING Function | The `ISMISSING` function tests whether a column of values is missing or null. For input column references, this function returns `true` or `false`. |
| ISNULL Function | The `ISNULL` function tests whether a column of values contains null values. For input column references, this function returns `true` or `false`. |

| | |
|---|---|
| NULL Function | The `NULL` function generates null values. |

**Source:**

Some source values that should match the State and Integer data types:

| State | Qty |
|-------|------|
| CA | 10 |
| OR | -10 |
| WA | 2.5 |
| ZZ | 15 |
| ID | |
| | 4 |

**Transformation:**

**Invalid State values**: You can test for invalid values for State using the following:

| | |
|---|---|
| **Transformation Name** | New formula |
| **Parameter: Formula type** | Single row formula |
| **Parameter: Formula** | ISMISMATCHED (State, 'State') |

The above transform flags rows 4 and 6 as mismatched.

> **NOTE:** A missing value is not valid for a type, including String type.

**Invalid Integer values:** You can test for valid matches for Qty using the following:

| | |
|---|---|
| **Transformation Name** | New formula |
| **Parameter: Formula type** | Single row formula |
| **Parameter: Formula** | (ISVALID (Qty, 'Integer') && (Qty > 0)) |
| **Parameter: New column name** | 'valid_Qty' |

The above transform flags as valid all rows where the `Qty` column is a valid integer that is greater than zero.

**Missing values:** The following transform tests for the presence of missing values in either column:

| | |
|---|---|
| **Transformation Name** | New formula |
| **Parameter: Formula type** | Single row formula |
| **Parameter: Formula** | (ISMISSING(State) \|\| ISMISSING(Qty)) |
| **Parameter: New column name** | 'missing_State_Qty' |

After re-organizing the columns using the `move` transform, the dataset should now look like the following:

| State | Qty | mismatched_State | valid_Qty | missing_State_Qty |
|-------|-----|------------------|-----------|-------------------|
| CA | 10 | false | true | false |
| OR | -10 | false | false | false |
| WA | 2.5 | false | false | false |
| ZZ | 15 | true | true | false |
| ID | | false | false | true |
| | 4 | false | true | true |

Since the data does not contain null values, the following transform generates null values based on the preceding criteria:

| Transformation Name | New formula |
|---------------------|-------------|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | `((mismatched_State == 'true') \|\| (valid_Qty == 'false') \|\| (missing_State_Qty == 'true')) ? NULL() : 'ok'` |
| Parameter: New column name | `'status'` |

You can then use the `ISNULL` check to remove the rows that fail the above test:

| Transformation Name | Filter rows |
|---------------------|-------------|
| Parameter: Condition | `Custom formula` |
| Parameter: Type of formula | `Custom single` |
| Parameter: Condition | `ISNULL('status')` |
| Parameter: Action | `Delete matching rows` |

**Results:**

Based on the above tests, the output dataset contains one row:

| State | Qty | mismatched_State | valid_Qty | missing_State_Qty | status |
|-------|-----|------------------|-----------|-------------------|--------|
| CA | 10 | false | true | false | ok |

# ISMISSING Function

The `ISMISSING` function tests whether a column of values is missing or null. For input column references, this function returns `true` or `false`.

- You can define a conditional test in a single step for valid values. See *IFMISSING Function*.
- Missing values are different from null values. To test for the presence of null values exclusively, see *ISNULL Function*.

**Wrangle vs. SQL:** This function is part of Wrangle , a proprietary data transformation language. Wrangle is not SQL. For more information, see *Wrangle Language*.

## Basic Usage

```
ismissing(Qty)
```

**Output:** Returns `true` if the value in the `Qty` column is missing.

## Syntax and Arguments

```
ismissing(column_string)
```

| Argument | Required? | Data Type | Description |
|---|---|---|---|
| column_string | Y | string | Name of column or string literal to be applied to the function |

For more information on syntax standards, see *Language Documentation Syntax Notes*.

**column_string**

Name of the column or string literal to be tested for missing values.

- Missing literals or column values generate missing string results.
- Multiple columns are supported.
- Wildcards are not supported.

**Usage Notes:**

| Required? | Data Type | Example Value |
|---|---|---|
| Yes | String literal or column reference | `myColumn` |

## Examples

> **Tip:** For additional examples, see *Common Tasks*.

**Example - Type check functions**

This example illustrates how various type checking functions can be applied to your data.

**Functions**:

| Item | Description |
|------|-------------|
| VALID Function | Tests whether a set of values is valid for a specified data type and is not a null value. |
| ISMISMATCHED Function | Tests whether a set of values is not valid for a specified data type. |
| ISMISSING Function | The `ISMISSING` function tests whether a column of values is missing or null. For input column references, this function returns `true` or `false`. |
| ISNULL Function | The `ISNULL` function tests whether a column of values contains null values. For input column references, this function returns `true` or `false`. |
| NULL Function | The `NULL` function generates null values. |

**Source:**

Some source values that should match the State and Integer data types:

| State | Qty |
|-------|-----|
| CA | 10 |
| OR | -10 |
| WA | 2.5 |
| ZZ | 15 |
| ID | |
| | 4 |

**Transformation:**

**Invalid State values**: You can test for invalid values for State using the following:

| | |
|--|--|
| **Transformation Name** | New formula |
| **Parameter: Formula type** | Single row formula |
| **Parameter: Formula** | ISMISMATCHED (State, 'State') |

The above transform flags rows 4 and 6 as mismatched.

> **NOTE:** A missing value is not valid for a type, including String type.

**Invalid Integer values:** You can test for valid matches for Qty using the following:

| | |
|--|--|
| **Transformation Name** | New formula |
| **Parameter: Formula type** | Single row formula |
| **Parameter: Formula** | (ISVALID (Qty, 'Integer') && (Qty > 0)) |
| **Parameter: New column name** | 'valid_Qty' |

The above transform flags as valid all rows where the `Qty` column is a valid integer that is greater than zero.

**Missing values:** The following transform tests for the presence of missing values in either column:

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | (ISMISSING(State) \|\| ISMISSING(Qty)) |
| Parameter: New column name | 'missing_State_Qty' |

After re-organizing the columns using the `move` transform, the dataset should now look like the following:

| State | Qty | mismatched_State | valid_Qty | missing_State_Qty |
|---|---|---|---|---|
| CA | 10 | false | true | false |
| OR | -10 | false | false | false |
| WA | 2.5 | false | false | false |
| ZZ | 15 | true | true | false |
| ID |  | false | false | true |
|  | 4 | false | true | true |

Since the data does not contain null values, the following transform generates null values based on the preceding criteria:

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | ((mismatched_State == 'true') \|\| (valid_Qty == 'false') \|\| (missing_State_Qty == 'true')) ? NULL() : 'ok' |
| Parameter: New column name | 'status' |

You can then use the `ISNULL` check to remove the rows that fail the above test:

| Transformation Name | Filter rows |
|---|---|
| Parameter: Condition | Custom formula |
| Parameter: Type of formula | Custom single |
| Parameter: Condition | ISNULL('status') |
| Parameter: Action | Delete matching rows |

**Results:**

Based on the above tests, the output dataset contains one row:

| State | Qty | mismatched_State | valid_Qty | missing_State_Qty | status |
|---|---|---|---|---|---|
| CA | 10 | false | true | false | ok |

# ISMISMATCHED Function

**Contents:**

---

Tests whether a set of values is not valid for a specified data type.

- For a tested value, this function returns `true` or `false` .
- Inputs can be literal values or column references.

You can define a conditional test in a single step for valid values. See *IFMISMATCHED Function.*

> **NOTE:** This function is similar to the `ISVALID` function, which tests for validity against a specified data type. However, unlike the `ISVALID` function, the `ISMISMATCHED` function also matches against missing values. See *VALID Function.*

**Wrangle vs. SQL:** This function is part of Wrangle , a proprietary data transformation language. Wrangle is not SQL. For more information, see *Wrangle Language.*

## Basic Usage

**Column reference example:**

```
ismismatched(Qty, 'Integer') || (Qty < 0)
```

**Output:** Returns `true` when the value in the `Qty` column does not contain a valid Integer and the value is less than zero.

**Numeric literal example:**

```
ismismatched('ZZ', 'State')
```

**Output:** Returns `true`, since the value `ZZ` is not a valid U.S. State code.

## Syntax and Arguments

```
ismismatched(column_string,datatype_literal)
```

| Argument | Required? | Data Type | Description |
|---|---|---|---|
| column_string | Y | string | Name of column or string literal to be applied to the function |
| datatype_literal | Y | string | String literal that identifies the data type against which to validate the source values |

For more information on syntax standards, see *Language Documentation Syntax Notes.*

**column_string**

Name of the column or string literal to be evaluated for mismatches against the specified type.

- Missing literals or column values generate missing string results.
  - Constants must be quoted (`'Hello, World'`).
- Multiple columns and wildcards are not supported.

**Usage Notes:**

| Required? | Data Type | Example Value |
|---|---|---|
| Yes | String literal or column reference | `myColumn` |

**datatype_literal**

Literal value for data type to which to validate the source column or string.

- Column references are not supported.

**Usage Notes:**

| Required? | Data Type | Example Value |
|---|---|---|
| Yes | String literal | `'Integer'` |

**Valid data type strings:**

When referencing a data type within a transform, you can use the following strings to identify each type:

> **NOTE:** In Wrangle transforms, these values are case-sensitive.

> **NOTE:** When specifying a data type by name, you must use the String value listed below. The Data Type value is the display name for the type.

| Data Type | String |
|---|---|
| String | `'String'` |
| Integer | `'Integer'` |
| Decimal | `'Float'` |
| Boolean | `'Bool'` |
| Social Security Number | `'SSN'` |
| Phone Number | `'Phone'` |
| Email Address | `'Emailaddress'` |
| Credit Card | `'Creditcard'` |
| Gender | `'Gender'` |

| Object | `'Map'` |
|---|---|
| Array | `'Array'` |
| IP Address | `'Ipaddress'` |
| URL | `'Url'` |
| HTTP Code | `'Httpcodes'` |
| Zip Code | `'Zipcode'` |
| State | `'State'` |
| Date / Time | `'Datetime'` |

## Examples

> **Tip:** For additional examples, see *Common Tasks*.

### Example - Type check functions

This example illustrates how various type checking functions can be applied to your data.

**Functions**:

| Item | Description |
|---|---|
| VALID Function | Tests whether a set of values is valid for a specified data type and is not a null value. |
| ISMISMATCHED Function | Tests whether a set of values is not valid for a specified data type. |
| ISMISSING Function | The `ISMISSING` function tests whether a column of values is missing or null. For input column references, this function returns `true` or `false`. |
| ISNULL Function | The `ISNULL` function tests whether a column of values contains null values. For input column references, this function returns `true` or `false`. |
| NULL Function | The `NULL` function generates null values. |

**Source:**

Some source values that should match the State and Integer data types:

| State | Qty |
|---|---|
| CA | 10 |
| OR | -10 |
| WA | 2.5 |
| ZZ | 15 |
| ID | |
| | 4 |

**Transformation:**

**Invalid State values**: You can test for invalid values for State using the following:

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | ISMISMATCHED (State, 'State') |

The above transform flags rows 4 and 6 as mismatched.

> **NOTE:** A missing value is not valid for a type, including String type.

**Invalid Integer values:** You can test for valid matches for Qty using the following:

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | (ISVALID (Qty, 'Integer') && (Qty > 0)) |
| Parameter: New column name | 'valid_Qty' |

The above transform flags as valid all rows where the `Qty` column is a valid integer that is greater than zero.

**Missing values:** The following transform tests for the presence of missing values in either column:

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | (ISMISSING(State) \|\| ISMISSING(Qty)) |
| Parameter: New column name | 'missing_State_Qty' |

After re-organizing the columns using the `move` transform, the dataset should now look like the following:

| State | Qty | mismatched_State | valid_Qty | missing_State_Qty |
|---|---|---|---|---|
| CA | 10 | false | true | false |
| OR | -10 | false | false | false |
| WA | 2.5 | false | false | false |
| ZZ | 15 | true | true | false |
| ID | | false | false | true |
| | 4 | false | true | true |

Since the data does not contain null values, the following transform generates null values based on the preceding criteria:

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: | ((mismatched_State == 'true') \|\| (valid_Qty == 'false') \|\| |

| | |
|---|---|
| **Formula** | `(missing_State_Qty == 'true')) ? NULL() : 'ok'` |
| **Parameter: New column name** | `'status'` |

You can then use the `ISNULL` check to remove the rows that fail the above test:

| | |
|---|---|
| **Transformation Name** | `Filter rows` |
| **Parameter: Condition** | `Custom formula` |
| **Parameter: Type of formula** | `Custom single` |
| **Parameter: Condition** | `ISNULL('status')` |
| **Parameter: Action** | `Delete matching rows` |

### Results:

Based on the above tests, the output dataset contains one row:

| State | Qty | mismatched_State | valid_Qty | missing_State_Qty | status |
|---|---|---|---|---|---|
| CA | 10 | false | true | false | ok |

# VALID Function

**Contents:**

- *Basic Usage*
- *Syntax and Arguments*
  - *column_string*
  - *datatype_literal*
- *Examples*
  - *Example - Type check functions*

---

Tests whether a set of values is valid for a specified data type and is not a null value.

- For a specified data type and set of values, this function returns `true` or `false`.
- Inputs can be literal values or column references.

You can use the `ISVALID` function keywords interchangeably.

- You can define a conditional test in a single step for valid values. See *IFVALID Function*.
- This function is similar to the `ISMISMATCHED` function, which tests for mismatches against a specified data type. However, the `ISMISMATCHED` function also matches against missing values, while the `ISVALID` function does not. See *ISMISMATCHED Function*.

**Wrangle vs. SQL:** This function is part of Wrangle , a proprietary data transformation language. Wrangle is not SQL. For more information, see *Wrangle Language*.

## Basic Usage

**Column reference example:**

```
(isvalid(Qty, 'Integer') && (Qty > 0))
```

**Output:** Returns `true` when the value in the `Qty` column contains a valid Integer and the value is greater than zero.

**Column reference example for Datetime:**

The Datetime data type requires a special formatting string as part of the evaluation of validity:

```
(isvalid(myDates, ['Datetime', 'yy-mm-dd hh:mm:ss','yyyy*mm*dd*HH:MM:SSX']))
```

**Output:** Returns `true` when the value in the `myDates` column conforms to either of the following date format strings:

```
yy-mm-dd hh:mm:ss
yyyy*mm*dd*HH:MM:SSX
```

For more information on these and other date format strings, see *DATEFORMAT Function*.

**Numeric literal example:**

```
isvalid(&apos;ZZ&apos;, &apos;State&apos;)
```

**Output:** Returns `false`, since the value `ZZ` is not a valid U.S. State code.

## Syntax and Arguments

```
isvalid(column_string,datatype_literal)
```

| Argument | Required? | Data Type | Description |
|---|---|---|---|
| column_string | Y | string | Name of column or string literal to be applied to the function |
| datatype_literal | Y | string | String literal that identifies the data type against which to validate the source values |

For more information on syntax standards, see *Language Documentation Syntax Notes*.

### column_string

Name of the column or string literal to be evaluated for validity.

- Missing literals or column values generate missing string results.
  - Constants must be quoted (`'Hello, World'`).
- Multiple columns and wildcards are not supported.

**Usage Notes:**

| Required? | Data Type | Example Value |
|---|---|---|
| Yes | String literal or column reference | `myColumn` |

### datatype_literal

Literal value for data type to which to match the source column or string. For more information, see *Valid Data Type Strings*.

- Column references are not supported.

**Usage Notes:**

| Required? | Data Type | Example Value |
|---|---|---|
| Yes | String literal | `'Integer'` |

**Valid data type strings:**

When referencing a data type within a transform, you can use the following strings to identify each type:

> **NOTE:** In Wrangle transforms, these values are case-sensitive.

> **NOTE:** When specifying a data type by name, you must use the String value listed below. The Data Type value is the display name for the type.

| Data Type | String |
|---|---|
| String | `'String'` |
| Integer | `'Integer'` |
| Decimal | `'Float'` |
| Boolean | `'Bool'` |
| Social Security Number | `'SSN'` |
| Phone Number | `'Phone'` |
| Email Address | `'Emailaddress'` |
| Credit Card | `'Creditcard'` |
| Gender | `'Gender'` |
| Object | `'Map'` |
| Array | `'Array'` |
| IP Address | `'Ipaddress'` |
| URL | `'Url'` |
| HTTP Code | `'Httpcodes'` |
| Zip Code | `'Zipcode'` |
| State | `'State'` |
| Date / Time | `'Datetime'` |

## Examples

> **Tip:** For additional examples, see *Common Tasks*.

**Example - Type check functions**

This example illustrates how various type checking functions can be applied to your data.

**Functions**:

| Item | Description |
|---|---|
| VALID Function | Tests whether a set of values is valid for a specified data type and is not a null value. |
| ISMISMATCHED Function | Tests whether a set of values is not valid for a specified data type. |
| ISMISSING Function | The `ISMISSING` function tests whether a column of values is missing or null. For input column references, this function returns `true` or `false`. |
| ISNULL Function | The `ISNULL` function tests whether a column of values contains null values. For input column references, this function returns `true` or `false`. |
| NULL Function | The `NULL` function generates null values. |

**Source:**

Some source values that should match the State and Integer data types:

| State | Qty |
|-------|-----|
| CA | 10 |
| OR | -10 |
| WA | 2.5 |
| ZZ | 15 |
| ID | |
| | 4 |

**Transformation:**

**Invalid State values**: You can test for invalid values for State using the following:

| Transformation Name | New formula |
|---------------------|-------------|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | ISMISMATCHED (State, 'State') |

The above transform flags rows 4 and 6 as mismatched.

> **NOTE:** A missing value is not valid for a type, including String type.

**Invalid Integer values:** You can test for valid matches for Qty using the following:

| Transformation Name | New formula |
|---------------------|-------------|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | (ISVALID (Qty, 'Integer') && (Qty > 0)) |
| Parameter: New column name | 'valid_Qty' |

The above transform flags as valid all rows where the `Qty` column is a valid integer that is greater than zero.

**Missing values:** The following transform tests for the presence of missing values in either column:

| Transformation Name | New formula |
|---------------------|-------------|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | (ISMISSING(State) \|\| ISMISSING(Qty)) |
| Parameter: New column name | 'missing_State_Qty' |

After re-organizing the columns using the `move` transform, the dataset should now look like the following:

| State | Qty | mismatched_State | valid_Qty | missing_State_Qty |
|-------|-----|------------------|-----------|-------------------|
| | | | | |

| | | | | |
|---|---|---|---|---|
| CA | 10 | false | true | false |
| OR | -10 | false | false | false |
| WA | 2.5 | false | false | false |
| ZZ | 15 | true | true | false |
| ID | | false | false | true |
| | 4 | false | true | true |

Since the data does not contain null values, the following transform generates null values based on the preceding criteria:

| | |
|---|---|
| **Transformation Name** | `New formula` |
| **Parameter: Formula type** | `Single row formula` |
| **Parameter: Formula** | `((mismatched_State == 'true') \|\| (valid_Qty == 'false') \|\| (missing_State_Qty == 'true')) ? NULL() : 'ok'` |
| **Parameter: New column name** | `'status'` |

You can then use the `ISNULL` check to remove the rows that fail the above test:

| | |
|---|---|
| **Transformation Name** | `Filter rows` |
| **Parameter: Condition** | `Custom formula` |
| **Parameter: Type of formula** | `Custom single` |
| **Parameter: Condition** | `ISNULL('status')` |
| **Parameter: Action** | `Delete matching rows` |

**Results:**

Based on the above tests, the output dataset contains one row:

| State | Qty | mismatched_State | valid_Qty | missing_State_Qty | status |
|---|---|---|---|---|---|
| CA | 10 | false | true | false | ok |

# PARSEINT Function

Evaluates a String input against the Integer datatype. If the input matches, the function outputs an Integer value. Input can be a literal, a column of values, or a function returning String values.

After you have converted your strings to integers, if a sufficient percentage of input strings from a column are successfully converted to the other date type, the column may be retyped.

**Wrangle vs. SQL:** This function is part of Wrangle , a proprietary data transformation language. Wrangle is not SQL. For more information, see *Wrangle Language*.

## Basic Usage

```
parseint(strInput)
```

**Output:** Returns the Integer data type value for `strInput` String values.

## Syntax and Arguments

```
parseint(str_input)
```

| Argument | Required? | Data Type | Description |
|----------|-----------|-----------|-------------|
| str_input | Y | String | Literal, name of a column, or a function returning String values to match |

For more information on syntax standards, see *Language Documentation Syntax Notes*.

**str_input**

Literal, column name, or function returning String values that are to be evaluated for conversion to Integer values.

- Missing values for this function in the source data result in null values in the output.
- Multiple columns and wildcards are not supported.

**Usage Notes:**

| Required? | Data Type | Example Value |
|-----------|-----------|---------------|
| Yes | String | `'5'` |

## Examples

> **Tip:** For additional examples, see *Common Tasks.*

## Example - type parsing functions

This example shows how to use parsing functions for evaluating input values against the function-specific data type.

**Functions**:

| Item | Description |
|---|---|
| PARSEBOOL Function | Evaluates a String input against the Boolean datatype. If the input matches, the function outputs a Boolean value. Input can be a literal, a column of values, or a function returning String values. |
| PARSEDATE Function | Evaluates an input against the default input formats or (if specified) an array of Datetime format strings in their listed order. If the input matches one of the formats, the function outputs a Datetime value. |
| PARSEFLOAT Function | Evaluates a String input against the Decimal datatype. If the input matches, the function outputs a Decimal value. Input can be a literal, a column of values, or a function returning String values. |
| PARSEINT Function | Evaluates a String input against the Integer datatype. If the input matches, the function outputs an Integer value. Input can be a literal, a column of values, or a function returning String values. |

**Source:**

The following table contains data on a series of races.

| raceId | disqualified | date | racerId | time_sc |
|---|---|---|---|---|
| 1 | FALSE | 2/1/20 | 1 | 24.22 |
| 2 | f | 2/8/20 | 1 | 25 |
| 3 | no | 2/8/20 | 1 | 24.11 |
| 4 | n | 1-Feb-20 | 2 | 26.1 |
| 5 | TRUE | 8-Feb-20 | 2.2 | -25.22 |
| 6 | t | 2/8/2020  10:16:00 AM | 2 | 25.44 |
| 7 | yes | 2/1/20 | 3 | 24 |
| 8 | y | 2/8/20 | 33 | 29.22 |
| 9 | 0 | 2/8/20 | 3 | 24.78 |
| 10 | 1 | 1-Feb-20 | 4 | 26.2.1 |
| 11 | FALSE | 8-Feb-20 | | 28.22 sec |
| 12 | FALSE | 2/8/2020  10:16:00 AM | 4 | 27.11 |

As you can see, this dataset has variation in values (FALSE, f, no, n) and problems with the data.

**Transformation:**

When the data is first imported, it may be properly typed for each column. To use the parsing functions, these columns should be converted to String data type:

| | |
|---|---|
| **Transformation Name** | Change column data type |
| **Parameter: Columns** | disqualified,date,racerId,time_sc |
| **Parameter: New type** | String |

Now, you can parse individual columns.

disqualified column:

| | |
|---|---|
| **Transformation Name** | `Edit column with formula` |
| **Parameter: Columns** | `disqualified` |
| **Parameter: Formula** | `PARSEBOOL($col)` |

racerId column:

| | |
|---|---|
| **Transformation Name** | `Edit column with formula` |
| **Parameter: Columns** | `racerId` |
| **Parameter: Formula** | `PARSEINT($col)` |

time_sc column:

| | |
|---|---|
| **Transformation Name** | `Edit column with formula` |
| **Parameter: Columns** | `time_sc` |
| **Parameter: Formula** | `PARSEFLOAT($col)` |

date column:

For the date column, the PARSEDATE function supports a default set of Datetime formats. Since some of the listed formats are different from these defaults, you must specify all of the formats. These formats are specified as an array of string values as the second argument of the function:

> **Tip:** For the PARSEDATE function, it's useful to use the Preview to verify that all of the dates in the column are represented in the array of output formats. You can see the available output formats through the data type menu at the top of a column in the Transformer Page.

| | |
|---|---|
| **Transformation Name** | `Edit column with formula` |
| **Parameter: Columns** | `date` |
| **Parameter: Formula** | `PARSEDATE($col, ['yyyy-MM-dd','yyyy\/MM\/dd','M\/d\/yyy hh:`<br>`mm','MMMM d, yyyy','MMM d, yyyy'])` |

After all of the date values have been standardized to the output format of the PARSEDATE function, you may choose to remove the time element of the values:

| | |
|---|---|
| **Transformation Name** | `Replace text or pattern` |
| **Parameter: Column** | `date` |
| **Parameter: Find** | `` ` {digit}{2}:{digit}{2}:{digit}{2}{end}` `` |
| **Parameter: Replace with** | `''` |

**Results:**

After executing the above steps, the data appears as follows. Notes on each column's output are below the table.

| raceId | disqualified | date | racerId | time_sc |
|--------|--------------|------|---------|---------|
| 1 | false | 2020-02-01 | 1 | 24.22 |
| 2 | false | 2020-02-08 | 1 | 25 |
| 3 | false | 2020-02-08 | 1 | 24.11 |
| 4 | false | 2020-02-01 | 2 | 26.1 |
| 5 | true | 2020-02-08 | *null* | -25.22 |
| 6 | true | 2020-02-08 | 2 | 25.44 |
| 7 | true | 2020-02-01 | 3 | 24 |
| 8 | true | 2020-02-08 | 33 | 29.22 |
| 9 | false | 2020-02-08 | 3 | 24.78 |
| 10 | true | 2020-02-01 | 4 | *null* |
| 11 | false | 2020-02-08 | *null* | *null* |
| 12 | false | 2020-02-08 | 4 | 27.11 |

disqualified column:

- The PARSEBOOL function normalizes all valid Boolean values to either `false` or `true`.

racerId column:

- The PARSEINT function writes invalid values as null values.
- The function writes empty values as null values.
- The value `33` remains, since it is a valid Integer. This value should be fixed manually.

time_sc:

- The PARSEFLOAT function writes the source value `25.00` as `25` in output.
- The source value `-25.22` remains. However, since this is time-based data, it needs to be fixed.
- Invalid values are written as nulls.

date column:

- All values are written in the standardized format: `yyyy-MM-dd HH:mm:ss`. Time data has been stripped.

# PARSEBOOL Function

Evaluates a String input against the Boolean datatype. If the input matches, the function outputs a Boolean value. Input can be a literal, a column of values, or a function returning String values.

After you have converted your strings values to Booleans, if a sufficient percentage of input strings from a column are successfully converted to the other date type, the column may be retyped.

**Wrangle vs. SQL:** This function is part of Wrangle , a proprietary data transformation language. Wrangle is not SQL. For more information, see *Wrangle Language*.

## Basic Usage

```
parsebool(strInput)
```

**Output:** Returns the Boolean data type value for `strInput` String values.

## Syntax and Arguments

```
parseint(str_input)
```

| Argument | Required? | Data Type | Description |
|----------|-----------|-----------|-------------|
| str_input | Y | String | Literal, name of a column, or a function returning String values to match |

For more information on syntax standards, see *Language Documentation Syntax Notes*.

**str_input**

Literal, column name, or function returning String values that are to be evaluated for conversion to Boolean values.

- Missing values for this function in the source data result in null values in the output.
- Multiple columns and wildcards are not supported.

**Usage Notes:**

| Required? | Data Type | Example Value |
|-----------|-----------|---------------|
| Yes | String | `'false'` |

## Examples

> **Tip:** For additional examples, see *Common Tasks*.

## Example - type parsing functions

This example shows how to use parsing functions for evaluating input values against the function-specific data type.

**Functions**:

| Item | Description |
|------|-------------|
| PARSEBOOL Function | Evaluates a String input against the Boolean datatype. If the input matches, the function outputs a Boolean value. Input can be a literal, a column of values, or a function returning String values. |
| PARSEDATE Function | Evaluates an input against the default input formats or (if specified) an array of Datetime format strings in their listed order. If the input matches one of the formats, the function outputs a Datetime value. |
| PARSEFLOAT Function | Evaluates a String input against the Decimal datatype. If the input matches, the function outputs a Decimal value. Input can be a literal, a column of values, or a function returning String values. |
| PARSEINT Function | Evaluates a String input against the Integer datatype. If the input matches, the function outputs an Integer value. Input can be a literal, a column of values, or a function returning String values. |

**Source:**

The following table contains data on a series of races.

| raceId | disqualified | date | racerId | time_sc |
|--------|--------------|------|---------|---------|
| 1 | FALSE | 2/1/20 | 1 | 24.22 |
| 2 | f | 2/8/20 | 1 | 25 |
| 3 | no | 2/8/20 | 1 | 24.11 |
| 4 | n | 1-Feb-20 | 2 | 26.1 |
| 5 | TRUE | 8-Feb-20 | 2.2 | -25.22 |
| 6 | t | 2/8/2020  10:16:00 AM | 2 | 25.44 |
| 7 | yes | 2/1/20 | 3 | 24 |
| 8 | y | 2/8/20 | 33 | 29.22 |
| 9 | 0 | 2/8/20 | 3 | 24.78 |
| 10 | 1 | 1-Feb-20 | 4 | 26.2.1 |
| 11 | FALSE | 8-Feb-20 | | 28.22 sec |
| 12 | FALSE | 2/8/2020  10:16:00 AM | 4 | 27.11 |

As you can see, this dataset has variation in values (FALSE, f, no, n) and problems with the data.

**Transformation:**

When the data is first imported, it may be properly typed for each column. To use the parsing functions, these columns should be converted to String data type:

| | |
|---|---|
| **Transformation Name** | `Change column data type` |
| **Parameter: Columns** | `disqualified,date,racerId,time_sc` |
| **Parameter: New type** | `String` |

Now, you can parse individual columns.

disqualified column:

| Transformation Name | Edit column with formula |
|---|---|
| Parameter: Columns | disqualified |
| Parameter: Formula | PARSEBOOL($col) |

racerId column:

| Transformation Name | Edit column with formula |
|---|---|
| Parameter: Columns | racerId |
| Parameter: Formula | PARSEINT($col) |

time_sc column:

| Transformation Name | Edit column with formula |
|---|---|
| Parameter: Columns | time_sc |
| Parameter: Formula | PARSEFLOAT($col) |

date column:

For the date column, the PARSEDATE function supports a default set of Datetime formats. Since some of the listed formats are different from these defaults, you must specify all of the formats. These formats are specified as an array of string values as the second argument of the function:

> **Tip:** For the PARSEDATE function, it's useful to use the Preview to verify that all of the dates in the column are represented in the array of output formats. You can see the available output formats through the data type menu at the top of a column in the Transformer Page.

| Transformation Name | Edit column with formula |
|---|---|
| Parameter: Columns | date |
| Parameter: Formula | PARSEDATE($col, ['yyyy-MM-dd','yyyy\/MM\/dd','M\/d\/yyy hh:mm','MMMM d, yyyy','MMM d, yyyy']) |

After all of the date values have been standardized to the output format of the PARSEDATE function, you may choose to remove the time element of the values:

| Transformation Name | Replace text or pattern |
|---|---|
| Parameter: Column | date |
| Parameter: Find | `` ` {digit}{2}:{digit}{2}:{digit}{2}{end}` `` |
| Parameter: Replace with | '' |

### Results:

After executing the above steps, the data appears as follows. Notes on each column's output are below the table.

| raceId | disqualified | date | racerId | time_sc |
|---|---|---|---|---|

| | | | | |
|---|---|---|---|---|
| 1 | false | 2020-02-01 | 1 | 24.22 |
| 2 | false | 2020-02-08 | 1 | 25 |
| 3 | false | 2020-02-08 | 1 | 24.11 |
| 4 | false | 2020-02-01 | 2 | 26.1 |
| 5 | true | 2020-02-08 | *null* | -25.22 |
| 6 | true | 2020-02-08 | 2 | 25.44 |
| 7 | true | 2020-02-01 | 3 | 24 |
| 8 | true | 2020-02-08 | 33 | 29.22 |
| 9 | false | 2020-02-08 | 3 | 24.78 |
| 10 | true | 2020-02-01 | 4 | *null* |
| 11 | false | 2020-02-08 | *null* | *null* |
| 12 | false | 2020-02-08 | 4 | 27.11 |

disqualified column:

- The PARSEBOOL function normalizes all valid Boolean values to either `false` or `true`.

racerId column:

- The PARSEINT function writes invalid values as null values.
- The function writes empty values as null values.
- The value `33` remains, since it is a valid Integer. This value should be fixed manually.

time_sc:

- The PARSEFLOAT function writes the source value `25.00` as `25` in output.
- The source value `-25.22` remains. However, since this is time-based data, it needs to be fixed.
- Invalid values are written as nulls.

date column:

- All values are written in the standardized format: `yyyy-MM-dd HH:mm:ss`. Time data has been stripped.

# PARSEFLOAT Function

Evaluates a String input against the Decimal datatype. If the input matches, the function outputs a Decimal value. Input can be a literal, a column of values, or a function returning String values.

After you have converted your strings values to decimals, if a sufficient percentage of input strings from a column are successfully converted to the other date type, the column may be retyped.

**Wrangle vs. SQL:** This function is part of Wrangle , a proprietary data transformation language. Wrangle is not SQL. For more information, see *Wrangle Language.*

## Basic Usage

```
parsefloat(strInput)
```

**Output:** Returns the Integer data type value for `strInput` String values.

## Syntax and Arguments

```
parsefloat(str_input)
```

| Argument | Required? | Data Type | Description |
|----------|-----------|-----------|-------------|
| str_input | Y | String | Literal, name of a column, or a function returning String values to match |

For more information on syntax standards, see *Language Documentation Syntax Notes.*

**str_input**

Literal, column name, or function returning String values that are to be evaluated for conversion to Decimal (Float) values.

- Missing values for this function in the source data result in null values in the output.
- Multiple columns and wildcards are not supported.

**Usage Notes:**

| Required? | Data Type | Example Value |
|-----------|-----------|---------------|
| Yes | String | `'5'` |

## Examples

> **Tip:** For additional examples, see *Common Tasks.*

## Example - type parsing functions

This example shows how to use parsing functions for evaluating input values against the function-specific data type.

**Functions**:

| Item | Description |
|------|-------------|
| PARSEBOOL Function | Evaluates a String input against the Boolean datatype. If the input matches, the function outputs a Boolean value. Input can be a literal, a column of values, or a function returning String values. |
| PARSEDATE Function | Evaluates an input against the default input formats or (if specified) an array of Datetime format strings in their listed order. If the input matches one of the formats, the function outputs a Datetime value. |
| PARSEFLOAT Function | Evaluates a String input against the Decimal datatype. If the input matches, the function outputs a Decimal value. Input can be a literal, a column of values, or a function returning String values. |
| PARSEINT Function | Evaluates a String input against the Integer datatype. If the input matches, the function outputs an Integer value. Input can be a literal, a column of values, or a function returning String values. |

**Source:**

The following table contains data on a series of races.

| raceId | disqualified | date | racerId | time_sc |
|--------|--------------|------|---------|---------|
| 1 | FALSE | 2/1/20 | 1 | 24.22 |
| 2 | f | 2/8/20 | 1 | 25 |
| 3 | no | 2/8/20 | 1 | 24.11 |
| 4 | n | 1-Feb-20 | 2 | 26.1 |
| 5 | TRUE | 8-Feb-20 | 2.2 | -25.22 |
| 6 | t | 2/8/2020  10:16:00 AM | 2 | 25.44 |
| 7 | yes | 2/1/20 | 3 | 24 |
| 8 | y | 2/8/20 | 33 | 29.22 |
| 9 | 0 | 2/8/20 | 3 | 24.78 |
| 10 | 1 | 1-Feb-20 | 4 | 26.2.1 |
| 11 | FALSE | 8-Feb-20 | | 28.22 sec |
| 12 | FALSE | 2/8/2020  10:16:00 AM | 4 | 27.11 |

As you can see, this dataset has variation in values (FALSE, f, no, n) and problems with the data.

**Transformation:**

When the data is first imported, it may be properly typed for each column. To use the parsing functions, these columns should be converted to String data type:

| | |
|------|------|
| **Transformation Name** | `Change column data type` |
| **Parameter: Columns** | `disqualified,date,racerId,time_sc` |
| **Parameter: New type** | `String` |

Now, you can parse individual columns.

disqualified column:

| Transformation Name | Edit column with formula |
|---|---|
| Parameter: Columns | disqualified |
| Parameter: Formula | PARSEBOOL($col) |

racerId column:

| Transformation Name | Edit column with formula |
|---|---|
| Parameter: Columns | racerId |
| Parameter: Formula | PARSEINT($col) |

time_sc column:

| Transformation Name | Edit column with formula |
|---|---|
| Parameter: Columns | time_sc |
| Parameter: Formula | PARSEFLOAT($col) |

date column:

For the date column, the PARSEDATE function supports a default set of Datetime formats. Since some of the listed formats are different from these defaults, you must specify all of the formats. These formats are specified as an array of string values as the second argument of the function:

> **Tip:** For the PARSEDATE function, it's useful to use the Preview to verify that all of the dates in the column are represented in the array of output formats. You can see the available output formats through the data type menu at the top of a column in the Transformer Page.

| Transformation Name | Edit column with formula |
|---|---|
| Parameter: Columns | date |
| Parameter: Formula | PARSEDATE($col, ['yyyy-MM-dd','yyyy\/MM\/dd','M\/d\/yyy hh:mm','MMMM d, yyyy','MMM d, yyyy']) |

After all of the date values have been standardized to the output format of the PARSEDATE function, you may choose to remove the time element of the values:

| Transformation Name | Replace text or pattern |
|---|---|
| Parameter: Column | date |
| Parameter: Find | `` ` {digit}{2}:{digit}{2}:{digit}{2}{end}` `` |
| Parameter: Replace with | '' |

### Results:

After executing the above steps, the data appears as follows. Notes on each column's output are below the table.

| raceId | disqualified | date | racerId | time_sc |
|---|---|---|---|---|

| | | | | |
|---|---|---|---|---|
| 1 | false | 2020-02-01 | 1 | 24.22 |
| 2 | false | 2020-02-08 | 1 | 25 |
| 3 | false | 2020-02-08 | 1 | 24.11 |
| 4 | false | 2020-02-01 | 2 | 26.1 |
| 5 | true | 2020-02-08 | *null* | -25.22 |
| 6 | true | 2020-02-08 | 2 | 25.44 |
| 7 | true | 2020-02-01 | 3 | 24 |
| 8 | true | 2020-02-08 | 33 | 29.22 |
| 9 | false | 2020-02-08 | 3 | 24.78 |
| 10 | true | 2020-02-01 | 4 | *null* |
| 11 | false | 2020-02-08 | *null* | *null* |
| 12 | false | 2020-02-08 | 4 | 27.11 |

disqualified column:

- The PARSEBOOL function normalizes all valid Boolean values to either `false` or `true`.

racerId column:

- The PARSEINT function writes invalid values as null values.
- The function writes empty values as null values.
- The value `33` remains, since it is a valid Integer. This value should be fixed manually.

time_sc:

- The PARSEFLOAT function writes the source value `25.00` as `25` in output.
- The source value `-25.22` remains. However, since this is time-based data, it needs to be fixed.
- Invalid values are written as nulls.

date column:

- All values are written in the standardized format: `yyyy-MM-dd HH:mm:ss`. Time data has been stripped.

# PARSEARRAY Function

Evaluates a String input against the Array datatype. If the input matches, the function outputs an Array value. Input can be a literal, a column of values, or a function returning String values.

After you have converted your strings to arrays, if a sufficient percentage of input strings from a column are successfully converted to the other data type, the column may be retyped.

> **Tip:** If the column is not automatically retyped as a result of this function, you can manually set the type to Array in a subsequent recipe step.

**Wrangle vs. SQL:** This function is part of Wrangle , a proprietary data transformation language. Wrangle is not SQL. For more information, see *Wrangle Language*.

## Basic Usage

```
parsearray(strInput)
```

**Output:** Returns the Array data type value for `strInput` String values.

## Syntax and Arguments

```
parsearray(str_input)
```

| Argument | Required? | Data Type | Description |
|----------|-----------|-----------|-------------|
| str_input | Y | String | Literal, name of a column, or a function returning String values to match |

For more information on syntax standards, see *Language Documentation Syntax Notes*.

### str_input

Literal, column name, or function returning String values that are to be evaluated for conversion to Array values.

- Missing values for this function in the source data result in null values in the output.
- Multiple columns and wildcards are not supported.

**Usage Notes:**

| Required? | Data Type | Example Value |
|-----------|-----------|---------------|
| Yes | String | `'[1,2,3]'` |

## Examples

> **Tip:** For additional examples, see *Common Tasks*.

**Example - parsing strings as an array**

**Source:**

The following table represents raw imported CSV data:

| setId | itemsA | itemsB |
|-------|--------|--------|
| s01 | "1,2,3" | 4 |
| s02 | "2,3,4" | 4 |
| s03 | "3,4,5" | 4 |
| s04 | "4,5,6" | 4 |
| s05 | "5,6,7" | 4 |
| s06 | "6,7,8" | 4 |

In the above table, you can see that the two items columns are interpreted differently. In the following steps, you can see how you can parse the data as array values before producing a new column intersecting the two arrays.

**Transformation:**

Create a new column to store the array version of `itemsA`:

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | itemsA |
| Parameter: New column name | arrA |

Remove the quotes from the column:

| Transformation Name | Replace text or pattern |
|---|---|
| Parameter: Column | arrA |
| Parameter: Find | `` `"` `` |
| Parameter: Replace with | '' |
| Parameter: Match all occurrences | true |

Now create the array by merging the array text value with square brackets and then using the PARSEARRAY function to evaluate the merged value as an array:

| Transformation Name | Edit column with formula |
|---|---|
| Parameter: Columns | arrA |
| Parameter: Formula | parsearray(merge(['[',arrA,']'])) |

You can create the second array column using a similar construction in a new column:

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | parsearray(merge(['[',itemsB,']'])) |
| Parameter: New column name | arrB |

Since both columns have been parsed as array values, you can use the ARRAYINTERSECT function to find the common values:

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | arrayintersect([arrA,arrB]) |
| Parameter: New column name | arrIntersectAB |

### Results:

| setId | itemsA | itemsB | arrA | arrB | arrIntersectAB |
|---|---|---|---|---|---|
| s01 | "1,2,3" | 4 | [1,2,3] | [4] | [] |
| s02 | "2,3,4" | 4 | [2,3,4] | [4] | [4] |
| s03 | "3,4,5" | 4 | [3,4,5] | [4] | [4] |
| s04 | "4,5,6" | 4 | [4,5,6] | [4] | [4] |
| s05 | "5,6,7" | 4 | [5,6,7] | [4] | [] |
| s06 | "6,7,8" | 4 | [6,7,8] | [4] | [] |

# PARSEOBJECT Function

Evaluates a String input against the Object datatype. If the input matches, the function outputs an Object value. Input can be a literal, a column of values, or a function returning String values.

After you have converted your strings to objects, if a sufficient percentage of input strings from a column are successfully converted to the other data type, the column may be retyped.

> **Tip:** If the column is not automatically retyped as a result of this function, you can manually set the type to Object in a subsequent recipe step.

**Wrangle vs. SQL:** This function is part of Wrangle , a proprietary data transformation language. Wrangle is not SQL. For more information, see *Wrangle Language*.

## Basic Usage

```
parseobject(strInput)
```

**Output:** Returns the Object data type value for `strInput` String values.

## Syntax and Arguments

```
parseobject(str_input)
```

| Argument | Required? | Data Type | Description |
|----------|-----------|-----------|-------------|
| str_input | Y | String | Literal, name of a column, or a function returning String values to match |

For more information on syntax standards, see *Language Documentation Syntax Notes*.

**str_input**

Literal, column name, or function returning String values that are to be evaluated for conversion to Object values.

- Missing values for this function in the source data result in null values in the output.
- Multiple columns and wildcards are not supported.

**Usage Notes:**

| Required? | Data Type | Example Value |
|-----------|-----------|---------------|
| Yes | String | `{"1","2","3"}` |

## Examples

## Example - parsing strings as objects

The following table shows a series of requests for inventory on three separate products. These are rolling requests, so inventory levels in the subsequent request are decreased based on the previous request.

| date | reqProdId | reqValue | prodA | prodB | prodC |
|------|-----------|----------|-------|-------|-------|
| 5/10/21 | prodA | 10 | 90 | 100 | 100 |
| 5/10/21 | prodC | 20 | 90 | 100 | 80 |
| 5/10/21 | prodA | 15 | 75 | 100 | 80 |
| 5/11/21 | prodB | 25 | 75 | 75 | 80 |
| 5/11/21 | prodA | 5 | 70 | 75 | 80 |
| 5/11/21 | prodC | 30 | 70 | 75 | 50 |
| 5/12/21 | prodB | 10 | 70 | 65 | 50 |

You must create a column containing the request information and the inventory level information for the requested product after the request has been fulfilled.

### Transformation:

The five data columns must be nested into an Object. The generated column is called `inventoryLevels`.

| Transformation Name | Nest columns into Objects |
|---------------------|---------------------------|
| Parameter: Columns | reqProdId,reqValue,prodA,prodB,prodC |
| Parameter: Nest columns to | Object |
| Parameter: New column name | inventoryLevels |

You can then build the inventory response column (`inventoryResponse`) using the FILTEROBJECT function:

| Transformation Name | New formula |
|---------------------|-------------|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | filterobject(parseobject(inventoryRequest), ['reqProdId','reqValue',reqProdId]) |
| Parameter: New column name | inventoryResponse |

### Results:

The `inventoryResponse` column contains the request information and the response information after the request has been fulfilled.

| date | reqProdId | reqValue | prodA | prodB | prodC | inventoryLevels | inventoryResponse |
|------|-----------|----------|-------|-------|-------|-----------------|-------------------|
| 5/10/21 | prodA | 10 | 90 | 100 | 100 | {"reqProdId":"prodA","reqValue":"10"," prodA":"90","prodB":"100","prodC":"100"} | {"reqProdId":"prodA"," reqValue":"10","prodA":" 90"} |

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 5/10/21 | prodC | 20 | 90 | 100 | 80 | {"reqProdId":"prodC","reqValue":"20"," prodA":"90","prodB":"100","prodC":"80"} | {"reqProdId":"prodC"," reqValue":"20","prodC":" 80"} |
| 5/10/21 | prodA | 15 | 75 | 100 | 80 | {"reqProdId":"prodA","reqValue":"15"," prodA":"75","prodB":"100","prodC":"80"} | {"reqProdId":"prodA"," reqValue":"15","prodA":" 75"} |
| 5/11/21 | prodB | 25 | 75 | 75 | 80 | {"reqProdId":"prodB","reqValue":"25"," prodA":"75","prodB":"75","prodC":"80"} | {"reqProdId":"prodB"," reqValue":"25","prodB":" 75"} |
| 5/11/21 | prodA | 5 | 70 | 75 | 80 | {"reqProdId":"prodA","reqValue":"5"," prodA":"70","prodB":"75","prodC":"80"} | {"reqProdId":"prodA"," reqValue":"5","prodA":" 70"} |
| 5/11/21 | prodC | 30 | 70 | 75 | 50 | {"reqProdId":"prodC","reqValue":"30"," prodA":"70","prodB":"75","prodC":"50"} | {"reqProdId":"prodC"," reqValue":"30","prodC":" 50"} |
| 5/12/21 | prodB | 10 | 70 | 65 | 50 | {"reqProdId":"prodB","reqValue":"10"," prodA":"70","prodB":"65","prodC":"50"} | {"reqProdId":"prodB"," reqValue":"10","prodB":" 65"} |

# PARSESTRING Function

Evaluates an input against the String datatype. If the input matches, the function outputs a String value. Input can be a literal, a column of values, or a function returning values. Values can be of any data type.

After you have converted your values to strings, if a sufficient percentage of inputs from a column are successfully converted to the other data type, the column may be retyped.

> **Tip:** If the column is not automatically retyped as a result of this function, you can manually set the type to String in a subsequent recipe step.

**Wrangle vs. SQL:** This function is part of Wrangle , a proprietary data transformation language. Wrangle is not SQL. For more information, see *Wrangle Language*.

## Basic Usage

```
parsestring(strInput)
```

**Output:** Returns the String data type value for `strInput` values.

## Syntax and Arguments

```
parsestring(str_input)
```

| Argument | Required? | Data Type | Description |
|----------|-----------|-----------|-------------|
| str_input | Y | any | Literal, name of a column, or a function returning values to match |

For more information on syntax standards, see *Language Documentation Syntax Notes*.

**str_input**

Literal, column name, or function returning values that are to be evaluated for conversion to String values.

- Missing values for this function in the source data result in null values in the output.
- Multiple columns and wildcards are not supported.

**Usage Notes:**

| Required? | Data Type | Example Values |
|-----------|-----------|----------------|
| Yes | any | 5 "Porsche" 3.4 |

## Examples

> **Tip:** For additional examples, see *Common Tasks*.

### Example - type parsing functions

**Source:**

The following table contains values for city, state, and zip code for locations in the United States:

| City | State | Zip |
|---|---|---|
| San Francisco | CA | 94105 |
| Seattle | WA | 98109 |
| Portland | OR | 97202 |
| San Diego | CA | 92109 |
| Brooklyn | NY | 11203 |
| Portland | ME | 4101 |
| Boston | MA | 2170 |

In the above table, you can see that some of the values are listed as four-digit zip codes, which are invalid. These values are likely to be interpreted as Integer values, which means that any leading zeroes are dropped. You can use the steps below to fix it.

**Transformation:**

Since you are working with integer values, you can use the following transformation to test the length of the values as if they were strings using the PARSESTRING function. If the values are only four characters long, then the value is merged with a leading `0`:

| Transformation Name | Edit column with formula |
|---|---|
| Parameter: Columns | Zip |
| Parameter: Formula | if(len(parsestring($col)) == 4, merge(['0',parsestring($col)]), $col) |

The `$col` reference points to the column that has been selected to be edited. In this case, that column is `Zip`. For more information, see *Source Metadata References*.

Depending on the number of rows in your dataset, the Designer Cloud application may not re-infer the data as Zip type. You can use the following transformation to change the data type for the column to Zip:

| Transformation Name | Change column data type |
|---|---|
| Parameter: Columns | Zip |
| Parameter: New type | Zipcode |

**Results:**

| City | State | Zip |
|---|---|---|
| San Francisco | CA | 94105 |

| | | |
|---|---|---|
| Seattle | WA | 98109 |
| Portland | OR | 97202 |
| San Diego | CA | 92109 |
| Brooklyn | NY | 11203 |
| Portland | ME | 04101 |
| Boston | MA | 02170 |

# Window Functions

Window functions apply to a subset of rows (a window) relative to the current row for computational purposes. These functions can be applied to the following transforms:

- *Window Transform*
- *Set Transform*
- *Derive Transform*

**Primary Key:**

When window functions are used, each row in the windowed output should be identified by a primary key. This **primary key**, which identifies unique rows, can be specified in one of the following ways:

- The `Order by` field indicates unique row identifiers. This field can contain one or more column references.
- A combination of `Order by` and `Group by` fields can create a unique identifier for each row.

This primary key requirement applies to window functions used with any supported transform.

> **NOTE:** Null values are ignored as inputs to these functions.

# PREV Function

**Contents:**

Extracts the value from a column that is a specified number of rows before the current value.

- The row from which to extract a value is determined by the order in which the rows are organized at the time that the function is executed.
- If you are working on a randomly generated sample of your dataset, the values that you see for this function might not correspond to the values that are generated on the full dataset during job execution.
- If the previous value is missing or null, this function generates a missing value.
- You can use the `group` and `order` parameters to define the groups of records and the order of those records to which this function is applied.
- This function works with the following transforms:
  - *Window Transform*
  - *Set Transform*
  - *Derive Transform*

**Wrangle vs. SQL:** This function is part of Wrangle , a proprietary data transformation language. Wrangle is not SQL. For more information, see *Wrangle Language*.

## Basic Usage

```
prev(myNumber, 1) order:Date
```

**Output:** Returns the value in the row in the `myNumber` column immediately preceding the current row, when ordered by `Date`.

## Syntax and Arguments

```
prev(col_ref, k_integer) order: order_col [group: group_col]
```

| Argument | Required? | Data Type | Description |
|---|---|---|---|
| col_ref | Y | string | Name of column whose values are applied to the function |
| k_integer | Y | integer (positive) | Number of rows before the current one from which to extract the value |

For more information on the `order` and `group` parameters, see *Window Transform*.

For more information on syntax standards, see *Language Documentation Syntax Notes*.

**col_ref**

Name of the column whose values are used to extract the value that is `k-integer` values before the current one.

- Multiple columns and wildcards are not supported.

**Usage Notes:**

| Required? | Data Type | Example Value |
|---|---|---|
| Yes | String (column reference) | `myColumn` |

### k_integer

Integer representing the number of rows before the current one from which to extract the value.

- Value must be a positive integer. For negative values, see *NEXT Function*.
- `k=1` represents the immediately preceding row value.
- If k is greater than or equal to the number of values in the column, all values in the generated column are missing. If a `group` parameter is applied, then this parameter should be no more than the maximum number of rows in the groups.
- If the range provided to the function exceeds the limits of the dataset, then the function generates a null value.
- If the range of the function is valid but includes missing values, the function generates a missing, non-null value.

**Usage Notes:**

| Required? | Data Type | Example Value |
|---|---|---|
| Yes | Integer | 4 |

## Examples

> **Tip:** For additional examples, see *Common Tasks*.

**Example - Examine prior order history**

This example describes how you can use the PREV function to analyze data that is available in a window in rows before the current one.

**Functions:**

| Item | Description |
|---|---|
| PREV Function | Extracts the value from a column that is a specified number of rows before the current value. |
| IF Function | The `IF` function allows you to build if/then/else conditional logic within your transforms. |

The following dataset contains orders for multiple customers over a period of a few days, listed in no particular order. You want to assess how order size has changed for each customer over time and to provide offers to your customers based on changes in order volume.

**Source:**

| Date | CustId | OrderId | OrderValue |
|---|---|---|---|
| | | | |

| | | | |
|---|---|---|---|
| 1/4/16 | C001 | Ord002 | 500 |
| 1/11/16 | C003 | Ord005 | 200 |
| 1/20/16 | C002 | Ord007 | 300 |
| 1/21/16 | C003 | Ord008 | 400 |
| 1/4/16 | C001 | Ord001 | 100 |
| 1/7/16 | C002 | Ord003 | 600 |
| 1/8/16 | C003 | Ord004 | 700 |
| 1/21/16 | C002 | Ord009 | 200 |
| 1/15/16 | C001 | Ord006 | 900 |

**Transformation:**

When the data is loaded into the Transformer page, you can use the `PREV` function to gather the order values for the previous two orders into a new column. The trick is to order the `window` transform by the date and group it by customer:

| | |
|---|---|
| **Transformation Name** | `Window` |
| **Parameter: Formulas** | `PREV(OrderValue, 1)` |
| **Parameter: Group by** | `CustId` |
| **Parameter: Order by** | `Date` |

| | |
|---|---|
| **Transformation Name** | `Window` |
| **Parameter: Formulas** | `PREV(OrderValue, 2)` |
| **Parameter: Group by** | `CustId` |
| **Parameter: Order by** | `Date` |

| | |
|---|---|
| **Transformation Name** | `Rename columns` |
| **Parameter: Option** | `Manual rename` |
| **Parameter: Column** | `window` |
| **Parameter: New column name** | `'OrderValue_1'` |

| | |
|---|---|
| **Transformation Name** | `Rename columns` |
| **Parameter: Option** | `Manual rename` |
| **Parameter: Column** | `window1` |
| **Parameter: New column name** | `'OrderValue_2'` |

You should now have the following columns in your dataset: `Date`, `CustId`, `OrderId`, `OrderValue`, `OrderValue_1`, `OrderValue_2`.

The two new columns represent the previous order and the order before that, respectively. Now, each row contains the current order (`OrderValue`) as well as the previous orders. Now, you want to take the following customer actions:

- If the current order is more than 20% greater than the sum of the two previous orders, send a rebate.
- If the current order is less than 90% of the sum of the two previous orders, send a coupon.
- Otherwise, send a holiday card.

To address the first one, you might add the following, which uses the `IF` function to test the value of the current order compared to the previous ones:

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | IF(OrderValue >= (1.2 * (OrderValue_1 + OrderValue_2)), 'send rebate', 'no action') |
| Parameter: New column name | 'CustomerAction' |

You can now see which customers are due a rebate. Now, edit the above and replace it with the following, which addresses the second condition. If neither condition is valid, then the result is `send holiday card`.

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | IF(OrderValue >= (1.2 * (OrderValue_1 + OrderValue_2)), 'send rebate', IF(OrderValue <= (1.2 * (OrderValue_1 + OrderValue_2)), 'send coupon', 'send holiday card')) |
| Parameter: New column name | 'CustomerAction' |

### Results:

After you delete the `OrderValue_1` and `OrderValue_2` columns, your dataset should look like the following. Since the transformations with `PREV` functions grouped by `CustId`, the order of records has changed.

| Date | CustId | OrderId | OrderValue | CustomerAction |
|---|---|---|---|---|
| 1/4/16 | C001 | Ord001 | 100 | send rebate |
| 1/7/16 | C001 | Ord002 | 500 | send rebate |
| 1/15/16 | C001 | Ord006 | 900 | send rebate |
| 1/8/16 | C003 | Ord004 | 700 | send rebate |
| 1/11/16 | C003 | Ord005 | 200 | send rebate |
| 1/21/16 | C003 | Ord008 | 400 | send coupon |
| 1/7/16 | C002 | Ord003 | 600 | send rebate |
| 1/20/16 | C002 | Ord007 | 300 | send rebate |
| 1/21/16 | C002 | Ord009 | 200 | send coupon |

# NEXT Function

**Contents:**

Extracts the value from a column that is a specified number of rows after the current value.

- The row from which to extract a value is determined by the order in which the rows are organized at the time that the function is executed.
- If you are working on a randomly generated sample of your dataset, the values that you see for this function might not correspond to the values that are generated on the full dataset during job execution.
- If the next value is missing or null, this function generates a missing value.
- You can use the `group` and `order` parameters to define the groups of records and the order of those records to which this function is applied.
- This function works with the following transforms:
    - *Window Transform*
    - *Set Transform*
    - *Derive Transform*

**Wrangle vs. SQL:** This function is part of Wrangle , a proprietary data transformation language. Wrangle is not SQL. For more information, see *Wrangle Language*.

## Basic Usage

```
next(myNumber, 1) order:Date
```

**Output:** Returns the value in the row in the `myNumber` column immediately after the current row when the dataset is ordered by `Date`.

## Syntax and Arguments

```
next(col_ref, k_integer) order: order_col [group: group_col]
```

| Argument | Required? | Data Type | Description |
|---|---|---|---|
| col_ref | Y | string | Name of column whose values are applied to the function |
| k_integer | Y | integer (positive) | Number of rows after the current one from which to extract the value |

For more information on the `order` and `group` parameters, see *Window Transform*.

For more information on syntax standards, see *Language Documentation Syntax Notes*.

**col_ref**

Name of the column whose values are used to extract the value that is `k-integer` values after the current one.

- Multiple columns and wildcards are not supported.

**Usage Notes:**

| Required? | Data Type | Example Value |
|---|---|---|
| Yes | String (column reference) | `myColumn` |

**k_integer**

Integer representing the number of rows after the current one from which to extract the value.

- Value must be a positive integer. For negative values, see *PREV Function*.
- `k=1` represents the immediately following row value.
- If k is greater than or equal to the number of values in the column, all values in the generated column are missing. If a `group` parameter is applied, then this parameter should be no more than the maximum number of rows in the groups.
- If the range provided to the function exceeds the limits of the dataset, then the function generates a null value.
- If the range of the function is valid but includes missing values, the function generates a missing, non-null value.

**Usage Notes:**

| Required? | Data Type | Example Value |
|---|---|---|
| Yes | Integer | 4 |

## Examples

> **Tip:** For additional examples, see *Common Tasks*.

**Example - Examine prior order history**

This example covers how to use the NEXT function to create windows of data from the current row and subsequent (next) rows in the dataset. You can then apply rolling computations across these windows of data.

**Functions:**

| Item | Description |
|---|---|
| NEXT Function | Extracts the value from a column that is a specified number of rows after the current value. |
| ROLLINGAVERRAGE Function | Computes the rolling average of values forward or backward of the current row within the specified column. |
| NUMFORMAT Function | Formats a numeric set of values according to the specified number formatting. Source values can be a literal numeric value, a function returning a numeric value, or reference to a column containing an Integer or Decimal values. |

**Source:**

The following dataset contains order information for the preceding 12 months. You want to compare the current month's average against the preceding quarter.

| Date | Amount |
|------|--------|
| 12/31/15 | 118 |
| 11/30/15 | 6 |
| 10/31/15 | 443 |
| 9/30/15 | 785 |
| 8/31/15 | 77 |
| 7/31/15 | 606 |
| 6/30/15 | 421 |
| 5/31/15 | 763 |
| 4/30/15 | 305 |
| 3/31/15 | 824 |
| 2/28/15 | 135 |
| 1/31/15 | 523 |

**Transformation:**

Using the `ROLLINGAVERAGE` function, you can generate a column containing the rolling average of the current month and the two previous months:

| Transformation Name | Window |
|---------------------|--------|
| Parameter: Formulas | `ROLLINGAVERAGE(Amount, 3, 0)` |
| Parameter: Order by | `-Date` |

Note the sign of the second parameter and the `order` parameter. The sort is in the reverse order of the `Date` parameter, which preserves the current sort order. As a result, the second parameter, which identifies the number of rows to use in the calculation, must be positive to capture the previous months.

Technically, this computation does not capture the prior quarter, since it includes the current quarter as part of the computation. You can use the following column to capture the rolling average of the preceding month, which then becomes the true rolling average for the prior quarter. The `window` column refers to the name of the column generated from the previous step:

| Transformation Name | Window |
|---------------------|--------|
| Parameter: Formulas | `NEXT(window, 1)` |
| Parameter: Order by | `-Date` |

Note that the order parameter must be preserved. This new column, `window1`, contains your prior quarter rolling average:

| Transformation Name | Rename columns |
|---------------------|----------------|
| Parameter: Option | Manual rename |
| | |

| Parameter: Column | `window1` |
|---|---|
| Parameter: New column name | `'Amount_PriorQtr'` |

You can reformat this numeric value:

| Transformation Name | `Edit column with formula` |
|---|---|
| Parameter: Columns | `Amount_PriorQtr` |
| Parameter: Formula | `NUMFORMAT(Amount_PriorQtr, '###.00')` |

You can use the following transformation to calculate the net change. This formula computes the change as a percentage of the prior quarter and then formats it as a two-digit percentage.

| Transformation Name | `New formula` |
|---|---|
| Parameter: Formula type | `Single row formula` |
| Parameter: Formula | `NUMFORMAT(((Amount - Amount_PriorQtr) / Amount_PriorQtr) * 100, '##.##')` |
| Parameter: New column name | `'NetChangePct_PriorQtr'` |

**Results:**

> **NOTE:** You might notice that there are computed values for `Amount_PriorQtr` for February and March. These values do not factor in a full three months because the data is not present. The January value does not exist since there is no data preceding it.

| Date | Amount | Amount_PriorQtr | NetChangePct_PriorQtr |
|---|---|---|---|
| 12/31/15 | 118 | 411.33 | -71.31 |
| 11/30/15 | 6 | 435.00 | -98.62 |
| 10/31/15 | 443 | 489.33 | -9.47 |
| 9/30/15 | 785 | 368.00 | 113.32 |
| 8/31/15 | 77 | 596.67 | -87.1 |
| 7/31/15 | 606 | 496.33 | 22.1 |
| 6/30/15 | 421 | 630.67 | -33.25 |
| 5/31/15 | 763 | 421.33 | 81.09 |
| 4/30/15 | 305 | 494.00 | -38.26 |
| 3/31/15 | 824 | 329.00 | 150.46 |
| 2/28/15 | 135 | 523.00 | -.74.19 |
| 1/31/15 | 523 | | |

# FILL Function

**Contents:**

---

Fills any missing or null values in the specified column with the most recent non-blank value, as determined by the specified window of rows before and after the blank value.

- The row from which to extract a value is determined by the order in which the rows are organized at the time that the function is executed.
- If you are working on a randomly generated sample of your dataset, the values that you see for this function might not correspond to the values that are generated on the full dataset during job execution.
- In addition to the column to which to apply the function, the function accepts two other parameters:
    - The first integer parameter defines the number of rows before the row being tested to scan for a non-empty value.
    - The second integer parameter defines the number of rows after the row being tested to scan for a non-empty value.
    - If no non-empty value is found within these rows, the empty value remains empty.
    - The default values are `-1` and `0` respectively, which performs an unlimited search before the row for a non-empty value to use to fill.

- You can use the `group` and `order` parameters to define the groups of records and the order of those records to which this function is applied.
- This function works with the following transforms:
    - *Window Transform*
    - *Set Transform*
    - *Derive Transform*

**Wrangle vs. SQL:** This function is part of Wrangle , a proprietary data transformation language. Wrangle is not SQL. For more information, see *Wrangle Language*.

## Basic Usage

```
fill(myNumber,-1,0)
```

**Output:** Returns all values from the `myNumber` column with any null cells filled by the most recent non-blank value.

```
fill(myNumber,-5,4)
```

**Output:** Returns all values from the `myNumber` column with any null cells filled by the most recent non-empty value within the window 5 rows before the current row and 4 rows after it.

## Syntax and Arguments

```
<span>fill</span><span>(col_ref, int_rows_before, int_rows_after) </span><span>order:
order_col [group: group_col]</span>
```

| Argument | Required? | Data Type | Description |
|---|---|---|---|
| col_ref | Y | string | Name of column whose values are applied to the function |
| int_rows_before | Y | integer | Number of rows before current row to scan for non-empty value. Default is $-1$, which scans all rows before. |
| int_rows_after | Y | integer | Number of rows after current row to scan for non-empty value. Default is $0$, which scans no rows after. |

For more information on the `order` and `group` parameters, see *Window Transform*.

For more information on syntax standards, see *Language Documentation Syntax Notes*.

### col_ref

Name of the column whose values are filled when null.

- Multiple columns and wildcards are not supported.

**Usage Notes:**

| Required? | Data Type | Example Value |
|---|---|---|
| Yes | String (column reference) | `myColumn` |

### int_rows_before

Number of rows before the current row to scan for the most recent non-empty value.

- Default value is `-1`, which scans all preceding rows.
- `0` does not scan before the current row.

**Usage Notes:**

| Required? | Data Type | Example Value |
|---|---|---|
| Yes | Integer | 5 |

### int_rows_after

Number of rows after the current row to scan for the most recent non-empty value.

- Default value is `0`, which does not scan rows after the current row.
- `-1` scans all following rows.

**Usage Notes:**

| Required? | Data Type | Example Value |
|---|---|---|
| Yes | Integer | 5 |

## Examples

### Example - Fill with quarterly forecast values

Your monthly sales data includes amount sold for each month. However, the source system only provides the quarterly forecast as part of the data for the first month of each quarter. You can use the `FILL` function to add the prior forecast to each month's data.

**Source:**

| Date | Amount | Forecast_Qtr |
|---|---|---|
| 1/31/15 | 523 | 1400 |
| 2/28/15 | 135 | |
| 3/31/15 | 824 | |
| 4/30/15 | 305 | 1500 |
| 5/31/15 | 763 | |
| 6/30/15 | 421 | |
| 7/31/15 | 606 | 1600 |
| 8/31/15 | 477 | |
| 9/30/15 | 785 | |
| 10/31/15 | 443 | 1700 |
| 11/30/15 | 622 | |
| 12/31/15 | 518 | |

**Transformation:**

You can use the following transformation to fill the prior forecast value for each blank month in the `Forecast_Qtr` column. Note that the `order` parameter must be set to `Date` to establish the proper sorting:

| | |
|---|---|
| Transformation Name | `Window` |
| Parameter: Formulas | `fill(Forecast_Qtr,-1,0)` |
| Parameter: Order by | `Date` |

You can now delete the `Forecast_Qtr` column and rename the generated `window` column to the deleted name.

To see how you are progressing each month, you might use the following transformation, which computes the average forecast for each month:

| | |
|---|---|
| Transformation Name | `New formula` |
| Parameter: Formula type | `Single row formula` |
| Parameter: Formula | `numformat((Forecast_Qtr/3),'####.##')` |
| Parameter: New column name | `'Forecast_Month_Avg'` |

You can then compare this value to the actual `Amount` value for each month:

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | numformat(((Amount - Forecast_Month_Avg)/Forecast_Month_Avg) *100, '##.00') |
| Parameter: New column name | 'MonthActualVForecast_Pct' |

**Results:**

| Date | Amount | Forecast_Qtr | Forecast_Month_Avg | MonthActualVForecast_Pct |
|---|---|---|---|---|
| 1/31/15 | 523 | 1400 | 466.67 | 12.07 |
| 2/28/15 | 135 | 1400 | 466.67 | -71.07 |
| 3/31/15 | 824 | 1400 | 466.67 | 76.57 |
| 4/30/15 | 305 | 1500 | 500 | -39.00 |
| 5/31/15 | 763 | 1500 | 500 | 52.60 |
| 6/30/15 | 421 | 1500 | 500 | -15.80 |
| 7/31/15 | 606 | 1600 | 533.33 | 13.63 |
| 8/31/15 | 477 | 1600 | 533.33 | -10.56 |
| 9/30/15 | 785 | 1600 | 533.33 | 47.19 |
| 10/31/15 | 443 | 1700 | 566.67 | -21.82 |
| 11/30/15 | 622 | 1700 | 566.67 | 9.76 |
| 12/31/15 | 518 | 1700 | 566.67 | -8.59 |

# RANK Function

Computes the rank of an ordered set of value within groups. Tie values are assigned the same rank, and the next ranking is incremented by the number of tie values.

- Rank values start at `1` and increment.
- Ranking order varies depending on the data type of the source data.
- You must use the `group` and `order` parameters to define the groups of records and the ordering column to which this function is applied.
- This function works with the following transforms:
    - *Window Transform*
    - *Set Transform*
    - *Derive Transform*
- This function assigns ranking values to match the total number of rows in a group. If needed, tie values can be assigned the same rank. For more information, see *DENSERANK Function*.

**Wrangle vs. SQL:** This function is part of Wrangle , a proprietary data transformation language. Wrangle is not SQL. For more information, see *Wrangle Language*.

## Basic Usage

```
rank()
```

**Output:** Generates the new column, which contains the ranking of `mySales`, grouped by the `Salesman` column.

## Syntax and Arguments

```
rank() order: order_col group: group_col
```

For more information on the `order` and `group` parameters, see *Window Transform*.

For more information on syntax standards, see *Language Documentation Syntax Notes*.

## Examples

> **Tip:** For additional examples, see *Common Tasks*.

**Example - Rank Functions**

This example demonstrates you to generate a ranked order of values.

**Functions**:

| Item | Description |
|---|---|
| RANK Function | Computes the rank of an ordered set of value within groups. Tie values are assigned the same rank, and the next ranking is incremented by the number of tie values. |
| DENSERANK Function | Computes the rank of an ordered set of value within groups. Tie values are assigned the same rank, and the next ranking is incremented by 1. |

**Source:**

The following dataset contains lap times for three racers in a four-lap race. Note that for some racers, there are tie values for lap times.

| Runner | Lap | Time |
|--------|-----|-------|
| Dave | 1 | 72.2 |
| Dave | 2 | 73.31 |
| Dave | 3 | 72.2 |
| Dave | 4 | 70.85 |
| Mark | 1 | 71.73 |
| Mark | 2 | 71.73 |
| Mark | 3 | 72.99 |
| Mark | 4 | 70.63 |
| Tom | 1 | 74.43 |
| Tom | 2 | 70.71 |
| Tom | 3 | 71.02 |
| Tom | 4 | 72.98 |

**Transformation:**

You can apply the `RANK()` function to the `Time` column, grouped by individual runner:

| | |
|--------|-------|
| **Transformation Name** | `Window` |
| **Parameter: Formulas** | `RANK()` |
| **Parameter: Group by** | `Runner` |
| **Parameter: Order by** | `Time` |

You can use the `DENSERANK()` function on the same column, grouping by runner:

| | |
|--------|-------|
| **Transformation Name** | `Window` |
| **Parameter: Formulas** | `DENSERANK()` |
| **Parameter: Group by** | `Runner` |
| **Parameter: Order by** | `Time` |

**Results:**

After renaming the columns, you have the following output:

| Runner | Lap | Time | Rank | Rank-Dense |
|--------|-----|-------|------|------------|
| Mark | 4 | 70.63 | 1 | 1 |
| Mark | 1 | 71.73 | 2 | 2 |
| Mark | 2 | 71.73 | 2 | 2 |
| Mark | 3 | 72.99 | 4 | 3 |
| Tom | 2 | 70.71 | 1 | 1 |

| | | | | |
|------|---|-------|---|---|
| Tom  | 3 | 71.02 | 2 | 2 |
| Tom  | 4 | 72.98 | 3 | 3 |
| Tom  | 1 | 74.43 | 4 | 4 |
| Dave | 4 | 70.85 | 1 | 1 |
| Dave | 1 | 72.2  | 2 | 2 |
| Dave | 3 | 72.2  | 2 | 2 |
| Dave | 2 | 73.31 | 4 | 3 |

# DENSERANK Function

Computes the rank of an ordered set of value within groups. Tie values are assigned the same rank, and the next ranking is incremented by 1.

- Rank values start at `1` and increment.
- Ranking order varies depending on the data type of the source data.
- You must use the `group` and `order` parameters to define the groups of records and the order of those records to which this function is applied.
- This function works with the following transforms:
    - *Window Transform*
    - *Set Transform*
    - *Derive Transform*
- This function assigns ranking of the next value of a set of ties as a single increment more. For more discrete ranking, see *RANK Function*.

**Wrangle vs. SQL:** This function is part of Wrangle , a proprietary data transformation language. Wrangle is not SQL. For more information, see *Wrangle Language*.

## Basic Usage

```
denserank() order:Times group:Racer
```

**Output:** Returns the dense ranking of `Times` values, grouped by the `Racer` column.

## Syntax and Arguments

```
denserank() order: order_col group: group_col
```

For more information on the `order` and `group` parameters, see *Window Transform*.

For more information on syntax standards, see *Language Documentation Syntax Notes*.

## Examples

> **Tip:** For additional examples, see *Common Tasks*.

**Example - Rank Functions**

This example demonstrates you to generate a ranked order of values.

**Functions**:

| Item | Description |
|------|-------------|
| RANK Function | Computes the rank of an ordered set of value within groups. Tie values are assigned the same rank, and the next ranking is incremented by the number of tie values. |
| DENSERANK Function | Computes the rank of an ordered set of value within groups. Tie values are assigned the same rank, and the next ranking is incremented by 1. |

**Source:**

The following dataset contains lap times for three racers in a four-lap race. Note that for some racers, there are tie values for lap times.

| Runner | Lap | Time |
|--------|-----|-------|
| Dave | 1 | 72.2 |
| Dave | 2 | 73.31 |
| Dave | 3 | 72.2 |
| Dave | 4 | 70.85 |
| Mark | 1 | 71.73 |
| Mark | 2 | 71.73 |
| Mark | 3 | 72.99 |
| Mark | 4 | 70.63 |
| Tom | 1 | 74.43 |
| Tom | 2 | 70.71 |
| Tom | 3 | 71.02 |
| Tom | 4 | 72.98 |

**Transformation:**

You can apply the `RANK()` function to the `Time` column, grouped by individual runner:

| | |
|---|---|
| **Transformation Name** | `Window` |
| **Parameter: Formulas** | `RANK()` |
| **Parameter: Group by** | `Runner` |
| **Parameter: Order by** | `Time` |

You can use the `DENSERANK()` function on the same column, grouping by runner:

| | |
|---|---|
| **Transformation Name** | `Window` |
| **Parameter: Formulas** | `DENSERANK()` |
| **Parameter: Group by** | `Runner` |
| **Parameter: Order by** | `Time` |

**Results:**

After renaming the columns, you have the following output:

| Runner | Lap | Time | Rank | Rank-Dense |
|--------|-----|-------|------|------------|
| Mark | 4 | 70.63 | 1 | 1 |
| Mark | 1 | 71.73 | 2 | 2 |
| Mark | 2 | 71.73 | 2 | 2 |
| Mark | 3 | 72.99 | 4 | 3 |
| Tom | 2 | 70.71 | 1 | 1 |

| | | | | |
|------|---|-------|---|---|
| Tom | 3 | 71.02 | 2 | 2 |
| Tom | 4 | 72.98 | 3 | 3 |
| Tom | 1 | 74.43 | 4 | 4 |
| Dave | 4 | 70.85 | 1 | 1 |
| Dave | 1 | 72.2 | 2 | 2 |
| Dave | 3 | 72.2 | 2 | 2 |
| Dave | 2 | 73.31 | 4 | 3 |

# ROLLINGAVERAGE Function

**Contents:**

---

Computes the rolling average of values forward or backward of the current row within the specified column.

- If an input value is missing or null, it is not factored in the computation. For example, for the first row in the dataset, the rolling average of previous values is the value in the first row.
- The row from which to extract a value is determined by the order in which the rows are organized based on the `order` parameter.
- If you are working on a randomly generated sample of your dataset, the values that you see for this function might not correspond to the values that are generated on the full dataset during job execution.
- The function takes a column name and two optional integer parameters that determine the window backward and forward of the current row.
    - The default integer parameter values are `-1` and `0`, which computes the rolling average from the current row back to the first row of the dataset.
- This function works with the Window transform. See *Window Transform*.

For more information on a non-rolling version of this function, see *AVERAGE Function*.

**Wrangle vs. SQL:** This function is part of Wrangle , a proprietary data transformation language. Wrangle is not SQL. For more information, see *Wrangle Language*.

## Basic Usage

**Column example:**

```
rollingaverage(myCol)
```

**Output:** Returns the rolling average of all values in the `myCol` column.

**Rows before example:**

```
rollingaverage(myNumber, 3)
```

**Output:** Returns the rolling average of the current row and the three previous row values in the `myNumber` column.

**Rows before and after example:**

```
rollingaverage(myNumber, 3, 2)
```

**Output:** Returns the rolling average of the three previous row values, the current row value, and the two rows after the current one in the `myNumber` column.

## Syntax and Arguments

```
rollingaverage(col_ref, rowsBefore_integer, rowsAfter_integer) order: order_col [group:
group_col]
```

| Argument | Required? | Data Type | Description |
|---|---|---|---|
| col_ref | Y | string | Name of column whose values are applied to the function |
| rowsBefore_integer | N | integer | Number of rows before the current one to include in the computation |
| rowsAfter_integer | N | integer | Number of rows after the current one to include in the computation |

For more information on the `order` and `group` parameters, see *Window Transform*.

For more information on syntax standards, see *Language Documentation Syntax Notes*.

### col_ref

Name of the column whose values are used to compute the rolling average.

- Multiple columns and wildcards are not supported.

**Usage Notes:**

| Required? | Data Type | Example Value |
|---|---|---|
| Yes | String (column reference to Integer or Decimal values) | `myColumn` |

### rowsBefore_integer, rowsAfter_integer

Integers representing the number of rows before or after the current one from which to compute the rolling average, including the current row. For example, if the first value is 5, the current row and the five rows before it are used in the computation. Negative values for `rowsAfter_integer` compute the rolling function from rows preceding the current one.

- `rowBefore=0` generates the current row value only.
- `rowBefore=-1` uses all rows preceding the current one.
- If `rowsAfter` is not specified, then the value `0` is applied.
- If a `group` parameter is applied, then these parameter values should be no more than the maximum number of rows in the groups.

**Usage Notes:**

| Required? | Data Type | Example Value |
|---|---|---|
| No | Integer | 4 |

## Examples

### Example - Compute prior quarter values

This example covers how to use the NEXT function to create windows of data from the current row and subsequent (next) rows in the dataset. You can then apply rolling computations across these windows of data.

**Functions:**

| Item | Description |
|------|-------------|
| NEXT Function | Extracts the value from a column that is a specified number of rows after the current value. |
| ROLLINGAVE RAGE Function | Computes the rolling average of values forward or backward of the current row within the specified column. |
| NUMFORMAT Function | Formats a numeric set of values according to the specified number formatting. Source values can be a literal numeric value, a function returning a numeric value, or reference to a column containing an Integer or Decimal values. |

**Source:**

The following dataset contains order information for the preceding 12 months. You want to compare the current month's average against the preceding quarter.

| Date | Amount |
|------|--------|
| 12/31/15 | 118 |
| 11/30/15 | 6 |
| 10/31/15 | 443 |
| 9/30/15 | 785 |
| 8/31/15 | 77 |
| 7/31/15 | 606 |
| 6/30/15 | 421 |
| 5/31/15 | 763 |
| 4/30/15 | 305 |
| 3/31/15 | 824 |
| 2/28/15 | 135 |
| 1/31/15 | 523 |

**Transformation:**

Using the `ROLLINGAVERAGE` function, you can generate a column containing the rolling average of the current month and the two previous months:

| Transformation Name | `Window` |
|---------------------|----------|
| Parameter: Formulas | `ROLLINGAVERAGE(Amount, 3, 0)` |
| Parameter: Order by | `-Date` |

Note the sign of the second parameter and the `order` parameter. The sort is in the reverse order of the `Date` parameter, which preserves the current sort order. As a result, the second parameter, which identifies the number of rows to use in the calculation, must be positive to capture the previous months.

Technically, this computation does not capture the prior quarter, since it includes the current quarter as part of the computation. You can use the following column to capture the rolling average of the preceding month, which then becomes the true rolling average for the prior quarter. The `window` column refers to the name of the column generated from the previous step:

| Transformation Name | Window |
|---|---|
| Parameter: Formulas | NEXT(window, 1) |
| Parameter: Order by | -Date |

Note that the order parameter must be preserved. This new column, `window1`, contains your prior quarter rolling average:

| Transformation Name | Rename columns |
|---|---|
| Parameter: Option | Manual rename |
| Parameter: Column | window1 |
| Parameter: New column name | 'Amount_PriorQtr' |

You can reformat this numeric value:

| Transformation Name | Edit column with formula |
|---|---|
| Parameter: Columns | Amount_PriorQtr |
| Parameter: Formula | NUMFORMAT(Amount_PriorQtr, '###.00') |

You can use the following transformation to calculate the net change. This formula computes the change as a percentage of the prior quarter and then formats it as a two-digit percentage.

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | NUMFORMAT(((Amount - Amount_PriorQtr) / Amount_PriorQtr) * 100, '##.##') |
| Parameter: New column name | 'NetChangePct_PriorQtr' |

**Results:**

> **NOTE:** You might notice that there are computed values for `Amount_PriorQtr` for February and March. These values do not factor in a full three months because the data is not present. The January value does not exist since there is no data preceding it.

| Date | Amount | Amount_PriorQtr | NetChangePct_PriorQtr |
|---|---|---|---|
|  |  |  |  |

| | | | |
|---|---|---|---|
| 12/31/15 | 118 | 411.33 | -71.31 |
| 11/30/15 | 6 | 435.00 | -98.62 |
| 10/31/15 | 443 | 489.33 | -9.47 |
| 9/30/15 | 785 | 368.00 | 113.32 |
| 8/31/15 | 77 | 596.67 | -87.1 |
| 7/31/15 | 606 | 496.33 | 22.1 |
| 6/30/15 | 421 | 630.67 | -33.25 |
| 5/31/15 | 763 | 421.33 | 81.09 |
| 4/30/15 | 305 | 494.00 | -38.26 |
| 3/31/15 | 824 | 329.00 | 150.46 |
| 2/28/15 | 135 | 523.00 | -.74.19 |
| 1/31/15 | 523 | | |

### Example - Rolling window functions

This example describes how to use rolling computational functions.

**Functions**:

| Item | Description |
|---|---|
| ROLLINGSUM Function | Computes the rolling sum of values forward or backward of the current row within the specified column. |
| ROLLINGAVERAGE Function | Computes the rolling average of values forward or backward of the current row within the specified column. |
| ROWNUMBER Function | Generates a new column containing the row number as sorted by the `order` parameter and optionally grouped by the `group` parameter. |

Also:

| Item | Description |
|---|---|
| MONTH Function | Derives the month integer value from a Datetime value. Source value can be a a reference to a column containing Datetime values or a literal. |
| FLOOR Function | Computes the largest integer that is not more than the input value. Input can be an Integer, a Decimal, a column reference, or an expression. |
| ROWNUMBER Function | Generates a new column containing the row number as sorted by the `order` parameter and optionally grouped by the `group` parameter. |

The following dataset contains sales data over the final quarter of the year.

**Source:**

| Date | Sales |
|---|---|
| 10/2/16 | 200 |
| 10/9/16 | 500 |
| 10/16/16 | 350 |
| 10/23/16 | 400 |
| 10/30/16 | 190 |

| | |
|---|---|
| 11/6/16 | 550 |
| 11/13/16 | 610 |
| 11/20/16 | 480 |
| 11/27/16 | 660 |
| 12/4/16 | 690 |
| 12/11/16 | 810 |
| 12/18/16 | 950 |
| 12/25/16 | 1020 |
| 1/1/17 | 680 |

**Transformation:**

First, you want to maintain the row information as a separate column. Since data is ordered already by the `Date` column, you can use the following:

| | |
|---|---|
| **Transformation Name** | `Window` |
| **Parameter: Formulas** | `ROWNUMBER()` |
| **Parameter: Order by** | `Date` |

Rename this column to `rowId` for week of quarter.

Now, you want to extract month and week information from the `Date` values. Deriving the month value:

| | |
|---|---|
| **Transformation Name** | `New formula` |
| **Parameter: Formula type** | `Single row formula` |
| **Parameter: Formula** | `MONTH(Date)` |
| **Parameter: New column name** | `'Month'` |

Deriving the quarter value:

| | |
|---|---|
| **Transformation Name** | `New formula` |
| **Parameter: Formula type** | `Single row formula` |
| **Parameter: Formula** | `(1 + FLOOR(((month-1)/3)))` |
| **Parameter: New column name** | `'QTR'` |

Deriving the week-of-quarter value:

| | |
|---|---|
| **Transformation Name** | `Window` |
| **Parameter: Formulas** | `ROWNUMBER()` |
| **Parameter: Group by** | `QTR` |
| **Parameter: Order by** | `Date` |

Rename this column `WOQ` (week of quarter).

Deriving the week-of-month value:

| | |
|---|---|
| Transformation Name | Window |
| Parameter: Formulas | ROWNUMBER() |
| Parameter: Group by | Month |
| Parameter: Order by | Date |

Rename this column `WOM` (week of month).

Now, you perform your rolling computations. Compute the running total of sales using the following:

| | |
|---|---|
| Transformation Name | Window |
| Parameter: Formulas | ROLLINGSUM(Sales, -1, 0) |
| Parameter: Group by | QTR |
| Parameter: Order by | Date |

The `-1` parameter is used in the above computation to gather the rolling sum of all rows of data from the current one to the first one. Note that the use of the `QTR` column for grouping, which moves the value for the `01/01/2017` into its own computational bucket. This may or may not be preferred.

Rename this column `QTD` (quarter to-date). Now, generate a similar column to compute the rolling average of weekly sales for the quarter:

| | |
|---|---|
| Transformation Name | Window |
| Parameter: Formulas | ROUND(ROLLINGAVERAGE(Sales, -1, 0)) |
| Parameter: Group by | QTR |
| Parameter: Order by | Date |

Since the `ROLLINGAVERAGE` function can compute fractional values, it is wrapped in the `ROUND` function for neatness. Rename this column `avgWeekByQuarter`.

**Results:**

When the unnecessary columns are dropped and some reordering is applied, your dataset should look like the following:

| Date | WOQ | Sales | QTD | avgWeekByQuarter |
|---|---|---|---|---|
| 10/2/16 | 1 | 200 | 200 | 200 |
| 10/9/16 | 2 | 500 | 700 | 350 |
| 10/16/16 | 3 | 350 | 1050 | 350 |
| 10/23/16 | 4 | 400 | 1450 | 363 |
| 10/30/16 | 5 | 190 | 1640 | 328 |
| 11/6/16 | 6 | 550 | 2190 | 365 |
| 11/13/16 | 7 | 610 | 2800 | 400 |

| 11/20/16 | 8 | 480 | 3280 | 410 |
|---|---|---|---|---|
| 11/27/16 | 9 | 660 | 3940 | 438 |
| 12/4/16 | 10 | 690 | 4630 | 463 |
| 12/11/16 | 11 | 810 | 5440 | 495 |
| 12/18/16 | 12 | 950 | 6390 | 533 |
| 12/25/16 | 13 | 1020 | 7410 | 570 |
| 1/1/17 | 1 | 680 | 680 | 680 |

## Example - Rolling computations for racing splits

This example describes how to use rolling statistical functions.

**Functions**:

| Item | Description |
|---|---|
| ROLLINGAVERAGE Function | Computes the rolling average of values forward or backward of the current row within the specified column. |
| ROLLINGMAX Function | Computes the rolling maximum of values forward or backward of the current row within the specified column. Inputs can be Integer, Decimal, or Datetime. |
| ROLLINGSTDEV Function | Computes the rolling standard deviation of values forward or backward of the current row within the specified column. |
| ROLLINGVAR Function | Computes the rolling variance of values forward or backward of the current row within the specified column. |
| ROLLINGSTDEVSAMP Function | Computes the rolling standard deviation of values forward or backward of the current row within the specified column using the sample statistical method. |
| ROLLINGVARSAMP Function | Computes the rolling variance of values forward or backward of the current row within the specified column using the sample statistical method. |

Also:

| Item | Description |
|---|---|
| MERGE Function | Merges two or more columns of String type to generate output of String type. Optionally, you can insert a delimiter between the merged values. |
| ROUND Function | Rounds input value to the nearest integer. Input can be an Integer, a Decimal, a column reference, or an expression. Optional second argument can be used to specify the number of digits to which to round. |

**Source:**

In this example, the following data comes from times recorded at regular intervals during a three-lap race around a track. The source data is in cumulative time in seconds (`time_sc`). You can use ROLLING and other windowing functions to break down the data into more meaningful metrics.

| lap | quarter | time_sc |
|---|---|---|
| 1 | 0 | 0.000 |
| 1 | 1 | 19.554 |
| 1 | 2 | 39.785 |
| 1 | 3 | 60.021 |
| | | |

| | | |
|---|---|---|
| 2 | 0 | 80.950 |
| 2 | 1 | 101.785 |
| 2 | 2 | 121.005 |
| 2 | 3 | 141.185 |
| 3 | 0 | 162.008 |
| 3 | 1 | 181.887 |
| 3 | 2 | 200.945 |
| 3 | 3 | 220.856 |

**Transformation:**

**Primary key:** Since the quarter information repeats every lap, there is no unique identifier for each row. The following steps create this identifier:

| | |
|---|---|
| **Transformation Name** | `Change column data type` |
| **Parameter: Columns** | `lap,quarter` |
| **Parameter: New type** | `String` |

| | |
|---|---|
| **Transformation Name** | `New formula` |
| **Parameter: Formula type** | `Single row formula` |
| **Parameter: Formula** | `MERGE(['l',lap,'q',quarter])` |
| **Parameter: New column name** | `'splitId'` |

**Get split times:** Use the following transform to break down the splits for each quarter of the race:

| | |
|---|---|
| **Transformation Name** | `New formula` |
| **Parameter: Formula type** | `Multiple row formula` |
| **Parameter: Formula** | `ROUND(time_sc - PREV(time_sc, 1), 3)` |
| **Parameter: Order rows by** | `splitId` |
| **Parameter: New column name** | `'split_time_sc'` |

**Compute rolling computations:** You can use the following types of computations to provide rolling metrics on the current and three previous splits:

| | |
|---|---|
| **Transformation Name** | `New formula` |
| **Parameter: Formula type** | `Multiple row formula` |
| **Parameter: Formula** | `ROLLINGAVERAGE(split_time_sc, 3)` |
| **Parameter: Order rows by** | `splitId` |
| **Parameter: New column name** | `'ravg'` |

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Multiple row formula |
| Parameter: Formula | ROLLINGMAX(split_time_sc, 3) |
| Parameter: Order rows by | splitId |
| Parameter: New column name | 'rmax' |

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Multiple row formula |
| Parameter: Formula | ROLLINGMIN(split_time_sc, 3) |
| Parameter: Order rows by | splitId |
| Parameter: New column name | 'rmin' |

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Multiple row formula |
| Parameter: Formula | ROUND(ROLLINGSTDEV(split_time_sc, 3), 3) |
| Parameter: Order rows by | splitId |
| Parameter: New column name | 'rstdev' |

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Multiple row formula |
| Parameter: Formula | ROUND(ROLLINGVAR(split_time_sc, 3), 3) |
| Parameter: Order rows by | splitId |
| Parameter: New column name | 'rvar' |

**Compute rolling computations using sample method:** These metrics compute the rolling STDEV and VAR on the current and three previous splits using the sample method:

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Multiple row formula |
| Parameter: Formula | ROUND(ROLLINGSTDEVSAMP(split_time_sc, 3), 3) |
| Parameter: Order rows by | splitId |
| Parameter: New column name | 'rstdev_samp' |

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Multiple row formula |

| Parameter: Formula | ROUND(ROLLINGVARSAMP(split_time_sc, 3), 3) |
|---|---|
| Parameter: Order rows by | splitId |
| Parameter: New column name | 'rvar_samp' |

**Results:**

When the above transforms have been completed, the results look like the following:

| lap | quarter | splitId | time_sc | split_time_sc | rvar_samp | rstdev_samp | rvar | rstdev | rmin | rmax | ravg |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 0 | l1q0 | 0 | | | | | | | | |
| 1 | 1 | l1q1 | 20.096 | 20.096 | | | 0 | 0 | 20.096 | 20.096 | 20.096 |
| 1 | 2 | l1q2 | 40.53 | 20.434 | 0.229 | 0.479 | 0.029 | 0.169 | 20.096 | 20.434 | 20.265 |
| 1 | 3 | l1q3 | 61.031 | 20.501 | 0.154 | 0.392 | 0.031 | 0.177 | 20.096 | 20.501 | 20.344 |
| 2 | 0 | l2q0 | 81.087 | 20.056 | 0.315 | 0.561 | 0.039 | 0.198 | 20.056 | 20.501 | 20.272 |
| 2 | 1 | l2q1 | 101.383 | 20.296 | 0.142 | 0.376 | 0.029 | 0.17 | 20.056 | 20.501 | 20.322 |
| 2 | 2 | l2q2 | 122.092 | 20.709 | 0.617 | 0.786 | 0.059 | 0.242 | 20.056 | 20.709 | 20.39 |
| 2 | 3 | l2q3 | 141.886 | 19.794 | 0.621 | 0.788 | 0.113 | 0.337 | 19.794 | 20.709 | 20.214 |
| 3 | 0 | l3q0 | 162.581 | 20.695 | 0.579 | 0.761 | 0.139 | 0.373 | 19.794 | 20.709 | 20.373 |
| 3 | 1 | l3q1 | 183.018 | 20.437 | 0.443 | 0.666 | 0.138 | 0.371 | 19.794 | 20.709 | 20.409 |
| 3 | 2 | l3q2 | 203.493 | 20.475 | 0.537 | 0.733 | 0.113 | 0.336 | 19.794 | 20.695 | 20.35 |
| 3 | 3 | l3q3 | 222.893 | 19.4 | 0.520 | 0.721 | 0.252 | 0.502 | 19.4 | 20.695 | 20.252 |

You can reduce the number of steps by applying a `window` transform such as the following:

| Transformation Name | Window |
|---|---|
| Parameter: Formula1 | lap |
| Parameter: Formula2 | rollingaverage(split_time_sc, 0, 3) |
| Parameter: Formula3 | rollingmax(split_time_sc, 0, 3) |
| Parameter: Formula4 | rollingmin(split_time_sc, 0, 3) |
| Parameter: Formula5 | round(rollingstdev(split_time_sc, 0, 3), 3) |
| Parameter: Formula6 | round(rollingvar(split_time_sc, 0, 3), 3) |
| Parameter: Formula7 | round(rollingstdevsamp(split_time_sc, 0, 3), 3) |
| Parameter: Formula8 | round(rollingvarsamp(split_time_sc, 0, 3), 3) |
| Parameter: Group by | lap |
| Parameter: Order by | lap |

However, you must rename all of the generated `windowX` columns.

# ROLLINGMODE Function

**Contents:**

Computes the rolling mode (most common value) forward or backward of the current row within the specified column. Input values can be Integer, Decimal, or Datetime data type.

- If an input value is missing or null, it is not factored in the computation. For example, for the first row in the dataset, the rolling mode of previous values is undefined.
- The row from which to extract a value is determined by the order in which the rows are organized based on the `order` parameter.
- If you are working on a randomly generated sample of your dataset, the values that you see for this function might not correspond to the values that are generated on the full dataset during job execution.
- The function takes a column name and two optional integer parameters that determine the window backward and forward of the current row.
    - The default integer parameter values are `-1` and `0`, which computes the rolling function from the current row back to the first row of the dataset.
- This function works with the Window transform. See *Window Transform*.

For more information on a non-rolling version of this function, see *MODE Function*.

**Wrangle vs. SQL:** This function is part of Wrangle , a proprietary data transformation language. Wrangle is not SQL. For more information, see *Wrangle Language*.

## Basic Usage

**Column example:**

```
rollingmode(myCol)
```

**Output:** Returns the rolling mode of all values in the `myCol` column.

**Rows before example:**

```
rollingmode(myNumber, 3)
```

**Output:** Returns the rolling mode of the current row and the three previous row values in the `myNumber` column.

**Rows before and after example:**

```
rollingmode(myNumber, 3, 2)
```

**Output:** Returns the rolling mode of the three previous row values, the current row value, and the two rows after the current one in the `myNumber` column.

## Syntax and Arguments

```
rollingmode(col_ref, rowsBefore_integer, rowsAfter_integer) order: order_col [group:
group_col]
```

| Argument | Required? | Data Type | Description |
|---|---|---|---|
| col_ref | Y | string | Name of column whose values are applied to the function |
| rowsBefore_integer | N | integer | Number of rows before the current one to include in the computation |
| rowsAfter_integer | N | integer | Number of rows after the current one to include in the computation |

For more information on the `order` and `group` parameters, see *Window Transform*.

For more information on syntax standards, see *Language Documentation Syntax Notes*.

**col_ref**

Name of the column whose values are used to compute the function.

> **NOTE:** If the input is in Datetime type, the output is in unixtime format. You can wrap these outputs in the DATEFORMAT function to generate the results in the appropriate Datetime format. See *DATEFORMAT Function*.

Multiple columns and wildcards are not supported.

**Usage Notes:**

| Required? | Data Type | Example Value |
|---|---|---|
| Yes | String (column reference to Integer or Decimal values) | `myColumn` |

**rowsBefore_integer, rowsAfter_integer**

Integers representing the number of rows before or after the current one from which to compute the rolling function, including the current row. For example, if the first value is 5, the current row and the five rows before it are used in the computation. Negative values for `rowsAfter_integer` compute the rolling function from rows preceding the current one.

- `rowBefore=0` generates the current row value only.
- `rowBefore=-1` uses all rows preceding the current one.
- If `rowsAfter` is not specified, then the value `0` is applied.
- If a `group` parameter is applied, then these parameter values should be no more than the maximum number of rows in the groups.

**Usage Notes:**

| Required? | Data Type | Example Value |
|---|---|---|
| No | Integer | 4 |

## Examples

> **Tip:** For additional examples, see *Common Tasks*.

### Example - Counting most common coin flips

In the following table, 20 coin flips are tabulated. You want to capture a rolling evaluation of the most common value.

**Source:**

| Turn | Result |
|------|--------|
| 1 | heads |
| 2 | heads |
| 3 | tails |
| 4 | heads |
| 5 | heads |
| 6 | tails |
| 7 | tails |
| 8 | heads |
| 9 | tails |
| 10 | heads |
| 11 | tails |
| 12 | heads |
| 13 | tails |
| 14 | heads |
| 15 | heads |
| 16 | tails |
| 17 | tails |
| 18 | tails |
| 19 | heads |
| 20 | heads |

**Transformation:**

To use the `ROLLINGMODE` function, the results need to be converted to numeric values:

| | |
|---|---|
| **Transformation Name** | `Edit column with formula` |
| **Parameter: Columns** | `Result` |
| **Parameter: Formula** | `if(Result == 'heads', 0, 1)` |

Now calculate the `ROLLINGMODE` for the preceding five values for each row:

| | |
|---|---|
| **Transformation Name** | `New formula` |

| Parameter: Formula type | Multiple row formula |
|---|---|
| Parameter: Formula | rollingmode(Result, 4, 0) |
| Parameter: Sort rows by | Turn |
| Parameter: New column name | 'mostCommonLast5' |

You can now convert the binary values back to text information:

| Transformation Name | Edit column with formula |
|---|---|
| Parameter: Columns | mostCommonLast5 |
| Parameter: Formula | if(mostCommonLast5 == 0, 'heads-last5', 'tails-last5') |

**Results:**

| Turn | Result | mostCommonLast5 |
|---|---|---|
| 1 | heads | heads-last5 |
| 2 | heads | heads-last5 |
| 3 | tails | heads-last5 |
| 4 | heads | heads-last5 |
| 5 | heads | heads-last5 |
| 6 | tails | heads-last5 |
| 7 | tails | tails-last5 |
| 8 | heads | heads-last5 |
| 9 | tails | tails-last5 |
| 10 | heads | tails-last5 |
| 11 | tails | tails-last5 |
| 12 | heads | heads-last5 |
| 13 | tails | tails-last5 |
| 14 | heads | heads-last5 |
| 15 | heads | heads-last5 |
| 16 | tails | heads-last5 |
| 17 | tails | tails-last5 |
| 18 | tails | tails-last5 |
| 19 | heads | tails-last5 |
| 20 | heads | tails-last5 |

# ROLLINGMAX Function

**Contents:**

Computes the rolling maximum of values forward or backward of the current row within the specified column. Inputs can be Integer, Decimal, or Datetime.

- If an input value is missing or null, it is not factored in the computation. For example, for the first row in the dataset, the rolling maximum of previous values is undefined.
- The row from which to extract a value is determined by the order in which the rows are organized based on the `order` parameter.
- If you are working on a randomly generated sample of your dataset, the values that you see for this function might not correspond to the values that are generated on the full dataset during job execution.
- The function takes a column name and two optional integer parameters that determine the window backward and forward of the current row.
    - The default integer parameter values are `-1` and `0`, which computes the rolling function from the current row back to the first row of the dataset.
- This function works with the Window transform. See *Window Transform*.

For more information on a non-rolling version of this function, see *MAX Function*.

**Wrangle vs. SQL:** This function is part of Wrangle , a proprietary data transformation language. Wrangle is not SQL. For more information, see *Wrangle Language*.

## Basic Usage

**Column example:**

```
rollingmax(myCol)
```

**Output:** Returns the rolling maximum of all values in the `myCol` column.

**Rows before example:**

```
rollingmax(myNumber, 3)
```

**Output:** Returns the rolling maximum of the current row and the three previous row values in the `myNumber` column.

**Rows before and after example:**

```
rollingmax(myNumber, 3, 2)
```

**Output:** Returns the rolling maximum of the three previous row values, the current row value, and the two rows after the current one in the `myNumber` column.

## Syntax and Arguments

```
rollingmax(col_ref, rowsBefore_integer, rowsAfter_integer) order: order_col [group:
group_col]
```

| Argument | Required? | Data Type | Description |
|---|---|---|---|
| col_ref | Y | string | Name of column whose values are applied to the function |
| rowsBefore_integer | N | integer | Number of rows before the current one to include in the computation |
| rowsAfter_integer | N | integer | Number of rows after the current one to include in the computation |

For more information on the `order` and `group` parameters, see *Window Transform*.

For more information on syntax standards, see *Language Documentation Syntax Notes*.

### col_ref

Name of the column whose values are used to compute the function. Inputs must be Integer, Decimal, or Datetime values.

> **NOTE:** If the input is in Datetime type, the output is in unixtime format. You can wrap these outputs in the DATEFORMAT function to generate the results in the appropriate Datetime format. See *DATEFORMAT Function*.

- Multiple columns and wildcards are not supported.

**Usage Notes:**

| Required? | Data Type | Example Value |
|---|---|---|
| Yes | String (column reference to Integer or Decimal values) | `myColumn` |

### rowsBefore_integer, rowsAfter_integer

Integers representing the number of rows before or after the current one from which to compute the rolling function, including the current row. For example, if the first value is `5`, the current row and the five rows before it are used in the computation. Negative values for `rowsAfter_integer` compute the rolling function from rows preceding the current one.

- `rowBefore=0` generates the current row value only.
- `rowBefore=-1` uses all rows preceding the current one.
- If `rowsAfter` is not specified, then the value `0` is applied.
- If a `group` parameter is applied, then these parameter values should be no more than the maximum number of rows in the groups.

**Usage Notes:**

| Required? | Data Type | Example Value |
|---|---|---|
| No | Integer | 4 |

## Examples

> **Tip:** For additional examples, see *Common Tasks*.

### Example - Rolling computations for racing splits

This example describes how to use rolling statistical functions.

**Functions**:

| Item | Description |
|---|---|
| ROLLINGAVERAGE Function | Computes the rolling average of values forward or backward of the current row within the specified column. |
| ROLLINGMAX Function | Computes the rolling maximum of values forward or backward of the current row within the specified column. Inputs can be Integer, Decimal, or Datetime. |
| ROLLINGSTDEV Function | Computes the rolling standard deviation of values forward or backward of the current row within the specified column. |
| ROLLINGVAR Function | Computes the rolling variance of values forward or backward of the current row within the specified column. |
| ROLLINGSTDEVSAMP Function | Computes the rolling standard deviation of values forward or backward of the current row within the specified column using the sample statistical method. |
| ROLLINGVARSAMP Function | Computes the rolling variance of values forward or backward of the current row within the specified column using the sample statistical method. |

Also:

| Item | Description |
|---|---|
| MERGE Function | Merges two or more columns of String type to generate output of String type. Optionally, you can insert a delimiter between the merged values. |
| ROUND Function | Rounds input value to the nearest integer. Input can be an Integer, a Decimal, a column reference, or an expression. Optional second argument can be used to specify the number of digits to which to round. |

**Source:**

In this example, the following data comes from times recorded at regular intervals during a three-lap race around a track. The source data is in cumulative time in seconds (`time_sc`). You can use ROLLING and other windowing functions to break down the data into more meaningful metrics.

| lap | quarter | time_sc |
|---|---|---|
| 1 | 0 | 0.000 |
| 1 | 1 | 19.554 |
| 1 | 2 | 39.785 |
| 1 | 3 | 60.021 |
| 2 | 0 | 80.950 |
| 2 | 1 | 101.785 |
| 2 | 2 | 121.005 |
| 2 | 3 | 141.185 |
| 3 | 0 | 162.008 |
| 3 | 1 | 181.887 |
| 3 | 2 | 200.945 |

| 3 | 3 | 220.856 |

**Transformation:**

**Primary key:** Since the quarter information repeats every lap, there is no unique identifier for each row. The following steps create this identifier:

| Transformation Name | Change column data type |
|---|---|
| Parameter: Columns | lap,quarter |
| Parameter: New type | String |

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | MERGE(['l',lap,'q',quarter]) |
| Parameter: New column name | 'splitId' |

**Get split times:** Use the following transform to break down the splits for each quarter of the race:

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Multiple row formula |
| Parameter: Formula | ROUND(time_sc - PREV(time_sc, 1), 3) |
| Parameter: Order rows by | splitId |
| Parameter: New column name | 'split_time_sc' |

**Compute rolling computations:** You can use the following types of computations to provide rolling metrics on the current and three previous splits:

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Multiple row formula |
| Parameter: Formula | ROLLINGAVERAGE(split_time_sc, 3) |
| Parameter: Order rows by | splitId |
| Parameter: New column name | 'ravg' |

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Multiple row formula |
| Parameter: Formula | ROLLINGMAX(split_time_sc, 3) |
| Parameter: Order rows by | splitId |
| Parameter: New column name | 'rmax' |

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Multiple row formula |
| Parameter: Formula | ROLLINGMIN(split_time_sc, 3) |
| Parameter: Order rows by | splitId |
| Parameter: New column name | 'rmin' |

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Multiple row formula |
| Parameter: Formula | ROUND(ROLLINGSTDEV(split_time_sc, 3), 3) |
| Parameter: Order rows by | splitId |
| Parameter: New column name | 'rstdev' |

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Multiple row formula |
| Parameter: Formula | ROUND(ROLLINGVAR(split_time_sc, 3), 3) |
| Parameter: Order rows by | splitId |
| Parameter: New column name | 'rvar' |

**Compute rolling computations using sample method:** These metrics compute the rolling STDEV and VAR on the current and three previous splits using the sample method:

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Multiple row formula |
| Parameter: Formula | ROUND(ROLLINGSTDEVSAMP(split_time_sc, 3), 3) |
| Parameter: Order rows by | splitId |
| Parameter: New column name | 'rstdev_samp' |

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Multiple row formula |
| Parameter: Formula | ROUND(ROLLINGVARSAMP(split_time_sc, 3), 3) |
| Parameter: Order rows by | splitId |
| Parameter: New column name | 'rvar_samp' |

**Results:**

When the above transforms have been completed, the results look like the following:

| lap | quarter | splitId | time_sc | split_time_sc | rvar_samp | rstdev_samp | rvar | rstdev | rmin | rmax | ravg |
|-----|---------|---------|---------|---------------|-----------|-------------|------|--------|------|------|------|
| 1 | 0 | l1q0 | 0 | | | | | | | | |
| 1 | 1 | l1q1 | 20.096 | 20.096 | | | 0 | 0 | 20.096 | 20.096 | 20.096 |
| 1 | 2 | l1q2 | 40.53 | 20.434 | 0.229 | 0.479 | 0.029 | 0.169 | 20.096 | 20.434 | 20.265 |
| 1 | 3 | l1q3 | 61.031 | 20.501 | 0.154 | 0.392 | 0.031 | 0.177 | 20.096 | 20.501 | 20.344 |
| 2 | 0 | l2q0 | 81.087 | 20.056 | 0.315 | 0.561 | 0.039 | 0.198 | 20.056 | 20.501 | 20.272 |
| 2 | 1 | l2q1 | 101.383 | 20.296 | 0.142 | 0.376 | 0.029 | 0.17 | 20.056 | 20.501 | 20.322 |
| 2 | 2 | l2q2 | 122.092 | 20.709 | 0.617 | 0.786 | 0.059 | 0.242 | 20.056 | 20.709 | 20.39 |
| 2 | 3 | l2q3 | 141.886 | 19.794 | 0.621 | 0.788 | 0.113 | 0.337 | 19.794 | 20.709 | 20.214 |
| 3 | 0 | l3q0 | 162.581 | 20.695 | 0.579 | 0.761 | 0.139 | 0.373 | 19.794 | 20.709 | 20.373 |
| 3 | 1 | l3q1 | 183.018 | 20.437 | 0.443 | 0.666 | 0.138 | 0.371 | 19.794 | 20.709 | 20.409 |
| 3 | 2 | l3q2 | 203.493 | 20.475 | 0.537 | 0.733 | 0.113 | 0.336 | 19.794 | 20.695 | 20.35 |
| 3 | 3 | l3q3 | 222.893 | 19.4 | 0.520 | 0.721 | 0.252 | 0.502 | 19.4 | 20.695 | 20.252 |

You can reduce the number of steps by applying a `window` transform such as the following:

| Transformation Name | `Window` |
|---------------------|----------|
| **Parameter: Formula1** | `lap` |
| **Parameter: Formula2** | `rollingaverage(split_time_sc, 0, 3)` |
| **Parameter: Formula3** | `rollingmax(split_time_sc, 0, 3)` |
| **Parameter: Formula4** | `rollingmin(split_time_sc, 0, 3)` |
| **Parameter: Formula5** | `round(rollingstdev(split_time_sc, 0, 3), 3)` |
| **Parameter: Formula6** | `round(rollingvar(split_time_sc, 0, 3), 3)` |
| **Parameter: Formula7** | `round(rollingstdevsamp(split_time_sc, 0, 3), 3)` |
| **Parameter: Formula8** | `round(rollingvarsamp(split_time_sc, 0, 3), 3)` |
| **Parameter: Group by** | `lap` |
| **Parameter: Order by** | `lap` |

However, you must rename all of the generated `windowX` columns.

# ROLLINGMIN Function

**Contents:**

Computes the rolling minimum of values forward or backward of the current row within the specified column. Input s can be Integer, Decimal, or Datetime.

- If an input value is missing or null, it is not factored in the computation. For example, for the first row in the dataset, the rolling minimum of previous values is undefined.
- The row from which to extract a value is determined by the order in which the rows are organized based on the `order` parameter.
- If you are working on a randomly generated sample of your dataset, the values that you see for this function might not correspond to the values that are generated on the full dataset during job execution.
- The function takes a column name and two optional integer parameters that determine the window backward and forward of the current row.
    - The default integer parameter values are `-1` and `0`, which computes the rolling function from the current row back to the first row of the dataset.
- This function works with the Window transform. See *Window Transform*.

For more information on a non-rolling version of this function, see *MIN Function*.

**Wrangle vs. SQL:** This function is part of Wrangle , a proprietary data transformation language. Wrangle is not SQL. For more information, see *Wrangle Language*.

## Basic Usage

**Column example:**

```
rollingmin(myCol)
```

**Output:** Returns the rolling minimum of all values in the `myCol` column.

**Rows before example:**

```
rollingmin(myNumber, 3)
```

**Output:** Returns the rolling minimum of the current row and the three previous row values in the `myNumber` column.

**Rows before and after example:**

```
rollingmin(myNumber, 3, 2)
```

**Output:** Returns the rolling minimum of the three previous row values, the current row value, and the two rows after the current one in the `myNumber` column.

## Syntax and Arguments

```
rollingmin(col_ref, rowsBefore_integer, rowsAfter_integer) order: order_col [group:
group_col]
```

| Argument | Required? | Data Type | Description |
|---|---|---|---|
| col_ref | Y | string | Name of column whose values are applied to the function |
| rowsBefore_integer | N | integer | Number of rows before the current one to include in the computation |
| rowsAfter_integer | N | integer | Number of rows after the current one to include in the computation |

For more information on the `order` and `group` parameters, see *Window Transform*.

For more information on syntax standards, see *Language Documentation Syntax Notes*.

### col_ref

Name of the column whose values are used to compute the function. Inputs must be Integer, Decimal, or Datetime values.

> **NOTE:** If the input is in Datetime type, the output is in unixtime format. You can wrap these outputs in the DATEFORMAT function to generate the results in the appropriate Datetime format. See *DATEFORMAT Function*.

- Multiple columns and wildcards are not supported.

**Usage Notes:**

| Required? | Data Type | Example Value |
|---|---|---|
| Yes | String (column reference to Integer or Decimal values) | `myColumn` |

### rowsBefore_integer, rowsAfter_integer

Integers representing the number of rows before or after the current one from which to compute the rolling function, including the current row. For example, if the first value is `5`, the current row and the five rows before it are used in the computation. Negative values for `rowsAfter_integer` compute the rolling function from rows preceding the current one.

- `rowBefore=0` generates the current row value only.
- `rowBefore=-1` uses all rows preceding the current one.
- If `rowsAfter` is not specified, then the value `0` is applied.
- If a `group` parameter is applied, then these parameter values should be no more than the maximum number of rows in the groups.

**Usage Notes:**

| Required? | Data Type | Example Value |
|---|---|---|
| No | Integer | 4 |

## Examples

> **Tip:** For additional examples, see *Common Tasks*.

### Example - Rolling computations for racing splits

This example describes how to use rolling statistical functions.

**Functions**:

| Item | Description |
|------|-------------|
| ROLLINGAVERAGE Function | Computes the rolling average of values forward or backward of the current row within the specified column. |
| ROLLINGMAX Function | Computes the rolling maximum of values forward or backward of the current row within the specified column. Inputs can be Integer, Decimal, or Datetime. |
| ROLLINGSTDEV Function | Computes the rolling standard deviation of values forward or backward of the current row within the specified column. |
| ROLLINGVAR Function | Computes the rolling variance of values forward or backward of the current row within the specified column. |
| ROLLINGSTDEVSAMP Function | Computes the rolling standard deviation of values forward or backward of the current row within the specified column using the sample statistical method. |
| ROLLINGVARSAMP Function | Computes the rolling variance of values forward or backward of the current row within the specified column using the sample statistical method. |

Also:

| Item | Description |
|------|-------------|
| MERGE Function | Merges two or more columns of String type to generate output of String type. Optionally, you can insert a delimiter between the merged values. |
| ROUND Function | Rounds input value to the nearest integer. Input can be an Integer, a Decimal, a column reference, or an expression. Optional second argument can be used to specify the number of digits to which to round. |

**Source:**

In this example, the following data comes from times recorded at regular intervals during a three-lap race around a track. The source data is in cumulative time in seconds (`time_sc`). You can use ROLLING and other windowing functions to break down the data into more meaningful metrics.

| lap | quarter | time_sc |
|-----|---------|---------|
| 1 | 0 | 0.000 |
| 1 | 1 | 19.554 |
| 1 | 2 | 39.785 |
| 1 | 3 | 60.021 |
| 2 | 0 | 80.950 |
| 2 | 1 | 101.785 |
| 2 | 2 | 121.005 |
| 2 | 3 | 141.185 |
| 3 | 0 | 162.008 |
| 3 | 1 | 181.887 |
| 3 | 2 | 200.945 |

| 3 | 3 | 220.856 |

**Transformation:**

**Primary key:** Since the quarter information repeats every lap, there is no unique identifier for each row. The following steps create this identifier:

| Transformation Name | Change column data type |
| --- | --- |
| Parameter: Columns | lap,quarter |
| Parameter: New type | String |

| Transformation Name | New formula |
| --- | --- |
| Parameter: Formula type | Single row formula |
| Parameter: Formula | MERGE(['l',lap,'q',quarter]) |
| Parameter: New column name | 'splitId' |

**Get split times:** Use the following transform to break down the splits for each quarter of the race:

| Transformation Name | New formula |
| --- | --- |
| Parameter: Formula type | Multiple row formula |
| Parameter: Formula | ROUND(time_sc - PREV(time_sc, 1), 3) |
| Parameter: Order rows by | splitId |
| Parameter: New column name | 'split_time_sc' |

**Compute rolling computations:** You can use the following types of computations to provide rolling metrics on the current and three previous splits:

| Transformation Name | New formula |
| --- | --- |
| Parameter: Formula type | Multiple row formula |
| Parameter: Formula | ROLLINGAVERAGE(split_time_sc, 3) |
| Parameter: Order rows by | splitId |
| Parameter: New column name | 'ravg' |

| Transformation Name | New formula |
| --- | --- |
| Parameter: Formula type | Multiple row formula |
| Parameter: Formula | ROLLINGMAX(split_time_sc, 3) |
| Parameter: Order rows by | splitId |
| Parameter: New column name | 'rmax' |

| | |
|---|---|
| **Transformation Name** | `New formula` |
| **Parameter: Formula type** | `Multiple row formula` |
| **Parameter: Formula** | `ROLLINGMIN(split_time_sc, 3)` |
| **Parameter: Order rows by** | `splitId` |
| **Parameter: New column name** | `'rmin'` |

| | |
|---|---|
| **Transformation Name** | `New formula` |
| **Parameter: Formula type** | `Multiple row formula` |
| **Parameter: Formula** | `ROUND(ROLLINGSTDEV(split_time_sc, 3), 3)` |
| **Parameter: Order rows by** | `splitId` |
| **Parameter: New column name** | `'rstdev'` |

| | |
|---|---|
| **Transformation Name** | `New formula` |
| **Parameter: Formula type** | `Multiple row formula` |
| **Parameter: Formula** | `ROUND(ROLLINGVAR(split_time_sc, 3), 3)` |
| **Parameter: Order rows by** | `splitId` |
| **Parameter: New column name** | `'rvar'` |

**Compute rolling computations using sample method:** These metrics compute the rolling STDEV and VAR on the current and three previous splits using the sample method:

| | |
|---|---|
| **Transformation Name** | `New formula` |
| **Parameter: Formula type** | `Multiple row formula` |
| **Parameter: Formula** | `ROUND(ROLLINGSTDEVSAMP(split_time_sc, 3), 3)` |
| **Parameter: Order rows by** | `splitId` |
| **Parameter: New column name** | `'rstdev_samp'` |

| | |
|---|---|
| **Transformation Name** | `New formula` |
| **Parameter: Formula type** | `Multiple row formula` |
| **Parameter: Formula** | `ROUND(ROLLINGVARSAMP(split_time_sc, 3), 3)` |
| **Parameter: Order rows by** | `splitId` |
| **Parameter: New column name** | `'rvar_samp'` |

**Results:**

When the above transforms have been completed, the results look like the following:

| lap | quarter | splitId | time_sc | split_time_sc | rvar_samp | rstdev_samp | rvar | rstdev | rmin | rmax | ravg |
|-----|---------|---------|---------|---------------|-----------|-------------|------|--------|------|------|------|
| 1 | 0 | l1q0 | 0 | | | | | | | | |
| 1 | 1 | l1q1 | 20.096 | 20.096 | | | 0 | 0 | 20.096 | 20.096 | 20.096 |
| 1 | 2 | l1q2 | 40.53 | 20.434 | 0.229 | 0.479 | 0.029 | 0.169 | 20.096 | 20.434 | 20.265 |
| 1 | 3 | l1q3 | 61.031 | 20.501 | 0.154 | 0.392 | 0.031 | 0.177 | 20.096 | 20.501 | 20.344 |
| 2 | 0 | l2q0 | 81.087 | 20.056 | 0.315 | 0.561 | 0.039 | 0.198 | 20.056 | 20.501 | 20.272 |
| 2 | 1 | l2q1 | 101.383 | 20.296 | 0.142 | 0.376 | 0.029 | 0.17 | 20.056 | 20.501 | 20.322 |
| 2 | 2 | l2q2 | 122.092 | 20.709 | 0.617 | 0.786 | 0.059 | 0.242 | 20.056 | 20.709 | 20.39 |
| 2 | 3 | l2q3 | 141.886 | 19.794 | 0.621 | 0.788 | 0.113 | 0.337 | 19.794 | 20.709 | 20.214 |
| 3 | 0 | l3q0 | 162.581 | 20.695 | 0.579 | 0.761 | 0.139 | 0.373 | 19.794 | 20.709 | 20.373 |
| 3 | 1 | l3q1 | 183.018 | 20.437 | 0.443 | 0.666 | 0.138 | 0.371 | 19.794 | 20.709 | 20.409 |
| 3 | 2 | l3q2 | 203.493 | 20.475 | 0.537 | 0.733 | 0.113 | 0.336 | 19.794 | 20.695 | 20.35 |
| 3 | 3 | l3q3 | 222.893 | 19.4 | 0.520 | 0.721 | 0.252 | 0.502 | 19.4 | 20.695 | 20.252 |

You can reduce the number of steps by applying a `window` transform such as the following:

| Transformation Name | Window |
|---------------------|--------|
| **Parameter: Formula1** | `lap` |
| **Parameter: Formula2** | `rollingaverage(split_time_sc, 0, 3)` |
| **Parameter: Formula3** | `rollingmax(split_time_sc, 0, 3)` |
| **Parameter: Formula4** | `rollingmin(split_time_sc, 0, 3)` |
| **Parameter: Formula5** | `round(rollingstdev(split_time_sc, 0, 3), 3)` |
| **Parameter: Formula6** | `round(rollingvar(split_time_sc, 0, 3), 3)` |
| **Parameter: Formula7** | `round(rollingstdevsamp(split_time_sc, 0, 3), 3)` |
| **Parameter: Formula8** | `round(rollingvarsamp(split_time_sc, 0, 3), 3)` |
| **Parameter: Group by** | `lap` |
| **Parameter: Order by** | `lap` |

However, you must rename all of the generated `windowX` columns.

# ROLLINGMODEDATE Function

**Contents:**

- *Basic Usage*
- *Syntax and Arguments*
    - *col_ref*
    - *rowsBefore_integer, rowsAfter_integer*
- *Examples*
    - *Example - Rolling date functions*

Computes the rolling mode (most common value) forward or backward of the current row within the specified column. Input values must be of Datetime data type.

- If an input value is missing or null, it is not factored in the computation. For example, for the first row in the dataset, the rolling mode of previous values is undefined.
- The row from which to extract a value is determined by the order in which the rows are organized based on the `order` parameter.
- If you are working on a randomly generated sample of your dataset, the values that you see for this function might not correspond to the values that are generated on the full dataset during job execution.
- The function takes a column name and two optional integer parameters that determine the window backward and forward of the current row.
    - The default integer parameter values are `-1` and `0`, which computes the rolling function from the current row back to the first row of the dataset.
- This function works with the Window transform. See *Window Transform*.

For more information on a non-rolling version of this function, see *MODEDATE Function*.

**Wrangle vs. SQL:** This function is part of Wrangle , a proprietary data transformation language. Wrangle is not SQL. For more information, see *Wrangle Language*.

## Basic Usage

**Column example:**

```
rollingmodedate(myDate)
```

**Output:** Returns the rolling mode of all values in the `myDate` column.

**Rows before example:**

```
rollingmodedate(myDate, 3)
```

**Output:** Returns the rolling mode of the current row and the three previous row values in the `myDate` column.

**Rows before and after example:**

```
rollingmodedate(myDate, 3, 2)
```

**Output:** Returns the rolling mode of the three previous row values, the current row value, and the two rows after the current one in the `myDate` column.

## Syntax and Arguments

```
rollingmodedate(col_ref, rowsBefore_integer, rowsAfter_integer) order: order_col [group:
group_col]
```

| Argument | Required? | Data Type | Description |
|---|---|---|---|
| col_ref | Y | string | Name of column whose values are applied to the function |
| rowsBefore_integer | N | integer | Number of rows before the current one to include in the computation |
| rowsAfter_integer | N | integer | Number of rows after the current one to include in the computation |

For more information on the `order` and `group` parameters, see *Window Transform*.

For more information on syntax standards, see *Language Documentation Syntax Notes*.

**col_ref**

Name of the column whose values are used to compute the function. Input values must be Datetime values.

Multiple columns and wildcards are not supported.

**Usage Notes:**

| Required? | Data Type | Example Value |
|---|---|---|
| Yes | String (column reference to Datetime values) | `transactionDate` |

**rowsBefore_integer, rowsAfter_integer**

Integers representing the number of rows before or after the current one from which to compute the rolling function, including the current row. For example, if the first value is `5`, the current row and the five rows before it are used in the computation. Negative values for `rowsAfter_integer` compute the rolling function from rows preceding the current one.

- `rowBefore=0` generates the current row value only.
- `rowBefore=-1` uses all rows preceding the current one.
- If `rowsAfter` is not specified, then the value `0` is applied.
- If a `group` parameter is applied, then these parameter values should be no more than the maximum number of rows in the groups.

**Usage Notes:**

| Required? | Data Type | Example Value |
|---|---|---|
| No | Integer | 4 |

## Examples

> **Tip:** For additional examples, see *Common Tasks*.

**Example - Rolling date functions**

This example describes how to use rolling functions for Datetime values.

**Functions**:

| Item | Description |
|---|---|
| ROLLINGMINDATE Function | Computes the rolling minimum of Date values forward or backward of the current row within the specified column. Inputs must be of Datetime type. |
| ROLLINGMAXDATE Function | Computes the rolling maximum of date values forward or backward of the current row within the specified column. Inputs must be of Datetime type. |
| ROLLINGMODEDATE Function | Computes the rolling mode (most common value) forward or backward of the current row within the specified column. Input values must be of Datetime data type. |

**Source:**

The following table contains an unordered list of orders:

| myDate | prodId | orderDollars |
|---|---|---|
| 2020-03-13 | p001 | 1445 |
| 2020-03-06 | p002 | 712 |
| 2020-03-16 | p003 | 1374 |
| 2020-03-23 | p001 | 1675 |
| 2020-04-09 | p002 | 1005 |
| 2020-08-09 | p003 | 984 |
| 2020-05-02 | p001 | 1395 |
| 2020-06-14 | p002 | 1866 |
| 2020-07-16 | p003 | 824 |
| 2020-09-02 | p001 | 1785 |
| 2020-08-31 | p002 | 697 |
| 2020-10-22 | p003 | 1513 |
| 2020-03-17 | p001 | 768 |
| 2020-03-21 | p002 | 1893 |
| 2020-03-23 | p003 | 1122 |
| 2020-04-06 | p001 | 805 |
| 2020-05-09 | p002 | 1752 |
| 2021-01-09 | p003 | 616 |
| 2020-08-18 | p001 | 1563 |
| 2020-09-12 | p002 | 730 |
| 2020-10-04 | p003 | 587 |
| 2021-02-15 | p001 | 1979 |
| 2021-02-22 | p002 | 134 |
| 2021-03-14 | p003 | 938 |

**Transformation:**

You can use the following Window transformation to calculate the rolling minimum, maximum, and mode dates for the last five orders for each product identifier:

| Transformation Name | `Window` |
| --- | --- |
| **Parameter: Formula1** | `ROLLINGMINDATE(orderDate, 4, 0)` |
| **Parameter: Formula2** | `ROLLINGMAXDATE(orderDate, 4, 0)` |
| **Parameter: Formula3** | `ROLLINGMODEDATE(orderDate, 4, 0)` |
| **Parameter: Group by** | `prodId` |
| **Parameter: Order by** | `prodId` |

You can use the following transformation to rename the generated `window` columns:

| Transformation Name | `Rename columns` |
| --- | --- |
| **Parameter: Option** | `Manual rename` |
| **Parameter: Column** | `window1` |
| **Parameter: New column name** | `rollingMinDate` |
| **Parameter: Parameter: Column** | `window2` |
| **Parameter: New column name** | `rollingMaxDate` |
| **Parameter: Parameter: Column** | `window3` |
| **Parameter: New column name** | `rollingModeDate` |

**Results:**

| orderDate | prodId | orderDollars | rollingMinDate | rollingMaxDate | rollingModeDate |
| --- | --- | --- | --- | --- | --- |
| 3/16/20 | p003 | 1374 | 3/16/20 | 3/16/20 | 3/16/20 |
| 8/9/20 | p003 | 984 | 3/16/20 | 8/9/20 | 3/16/20 |
| 7/16/20 | p003 | 824 | 3/16/20 | 8/9/20 | 3/16/20 |
| 10/22/20 | p003 | 1513 | 3/16/20 | 10/22/20 | 3/16/20 |
| 3/23/20 | p003 | 1122 | 3/16/20 | 10/22/20 | 3/16/20 |
| 1/9/21 | p003 | 616 | 3/23/20 | 1/9/21 | 3/23/20 |
| 10/4/20 | p003 | 587 | 3/23/20 | 1/9/21 | 3/23/20 |
| 3/14/21 | p003 | 938 | 3/23/20 | 3/14/21 | 3/23/20 |
| 3/13/20 | p001 | 1445 | 3/13/20 | 3/13/20 | 3/13/20 |
| 3/23/20 | p001 | 1675 | 3/13/20 | 3/23/20 | 3/13/20 |
| 5/2/20 | p001 | 1395 | 3/13/20 | 5/2/20 | 3/13/20 |
| 9/2/20 | p001 | 1785 | 3/13/20 | 9/2/20 | 3/13/20 |
| 3/17/20 | p001 | 768 | 3/13/20 | 9/2/20 | 3/13/20 |
| 4/6/20 | p001 | 805 | 3/17/20 | 9/2/20 | 3/17/20 |
| 8/18/20 | p001 | 1563 | 3/17/20 | 9/2/20 | 3/17/20 |

| | | | | | |
|---------|------|------|---------|---------|---------|
| 2/15/21 | p001 | 1979 | 3/17/20 | 2/15/21 | 3/17/20 |
| 3/6/20  | p002 | 712  | 3/6/20  | 3/6/20  | 3/6/20  |
| 4/9/20  | p002 | 1005 | 3/6/20  | 4/9/20  | 3/6/20  |
| 6/14/20 | p002 | 1866 | 3/6/20  | 6/14/20 | 3/6/20  |
| 8/31/20 | p002 | 697  | 3/6/20  | 8/31/20 | 3/6/20  |
| 3/21/20 | p002 | 1893 | 3/6/20  | 8/31/20 | 3/6/20  |
| 5/9/20  | p002 | 1752 | 3/21/20 | 8/31/20 | 3/21/20 |
| 9/12/20 | p002 | 730  | 3/21/20 | 9/12/20 | 3/21/20 |
| 2/22/21 | p002 | 134  | 3/21/20 | 2/22/21 | 3/21/20 |

# ROLLINGMAXDATE Function

**Contents:**

Computes the rolling maximum of date values forward or backward of the current row within the specified column. Inputs must be of Datetime type.

- If an input value is missing or null, it is not factored in the computation. For example, for the first row in the dataset, the rolling maximum of previous values is undefined.
- The row from which to extract a value is determined by the order in which the rows are organized based on the `order` parameter.
- If you are working on a randomly generated sample of your dataset, the values that you see for this function might not correspond to the values that are generated on the full dataset during job execution.
- The function takes a column name and two optional integer parameters that determine the window backward and forward of the current row.
    - The default integer parameter values are `-1` and `0`, which computes the rolling function from the current row back to the first row of the dataset.
- This function works with the Window transform. See *Window Transform*.

For more information on a non-rolling version of this function, see *MAXDATE Function*.

**Wrangle vs. SQL:** This function is part of Wrangle , a proprietary data transformation language. Wrangle is not SQL. For more information, see *Wrangle Language*.

## Basic Usage

**Column example:**

```
rollingmaxdate(<span>myDate</span>)
```

**Output:** Returns the rolling maximum of all values in the `myDate` column.

**Rows before example:**

```
rollingmaxdate(<span>myDate</span>, 3)
```

**Output:** Returns the rolling maximum of the current row and the three previous row values in the `myDate` column.

**Rows before and after example:**

```
rollingmaxdate(myDate, 3, 2)
```

**Output:** Returns the rolling maximum of the three previous row values, the current row value, and the two rows after the current one in the `myDate` column.

## Syntax and Arguments

```
rollingmaxdate(col_ref, rowsBefore_integer, rowsAfter_integer) order: order_col [group:
group_col]
```

| Argument | Required? | Data Type | Description |
|---|---|---|---|
| col_ref | Y | string | Name of column whose values are applied to the function |
| rowsBefore_integer | N | integer | Number of rows before the current one to include in the computation |
| rowsAfter_integer | N | integer | Number of rows after the current one to include in the computation |

For more information on the `order` and `group` parameters, see *Window Transform*.

For more information on syntax standards, see *Language Documentation Syntax Notes*.

**col_ref**

Name of the column whose values are used to compute the function. Inputs must be Datetime values.

Multiple columns and wildcards are not supported.

**Usage Notes:**

| Required? | Data Type | Example Value |
|---|---|---|
| Yes | String (column reference to Datetime values) | `transactionDate` |

**rowsBefore_integer, rowsAfter_integer**

Integers representing the number of rows before or after the current one from which to compute the rolling function, including the current row. For example, if the first value is `5`, the current row and the five rows before it are used in the computation. Negative values for `rowsAfter_integer` compute the rolling function from rows preceding the current one.

- `rowBefore=0` generates the current row value only.
- `rowBefore=-1` uses all rows preceding the current one.
- If `rowsAfter` is not specified, then the value `0` is applied.
- If a `group` parameter is applied, then these parameter values should be no more than the maximum number of rows in the groups.

**Usage Notes:**

| Required? | Data Type | Example Value |
|---|---|---|
| No | Integer | 4 |

Examples

> **Tip:** For additional examples, see *Common Tasks*.

**Example - Rolling date functions**

This example describes how to use rolling functions for Datetime values.

**Functions**:

| Item | Description |
|---|---|
| ROLLINGMINDATE Function | Computes the rolling minimum of Date values forward or backward of the current row within the specified column. Inputs must be of Datetime type. |
| ROLLINGMAXDATE Function | Computes the rolling maximum of date values forward or backward of the current row within the specified column. Inputs must be of Datetime type. |
| ROLLINGMODEDATE Function | Computes the rolling mode (most common value) forward or backward of the current row within the specified column. Input values must be of Datetime data type. |

### Source:

The following table contains an unordered list of orders:

| myDate | prodId | orderDollars |
|---|---|---|
| 2020-03-13 | p001 | 1445 |
| 2020-03-06 | p002 | 712 |
| 2020-03-16 | p003 | 1374 |
| 2020-03-23 | p001 | 1675 |
| 2020-04-09 | p002 | 1005 |
| 2020-08-09 | p003 | 984 |
| 2020-05-02 | p001 | 1395 |
| 2020-06-14 | p002 | 1866 |
| 2020-07-16 | p003 | 824 |
| 2020-09-02 | p001 | 1785 |
| 2020-08-31 | p002 | 697 |
| 2020-10-22 | p003 | 1513 |
| 2020-03-17 | p001 | 768 |
| 2020-03-21 | p002 | 1893 |
| 2020-03-23 | p003 | 1122 |
| 2020-04-06 | p001 | 805 |
| 2020-05-09 | p002 | 1752 |
| 2021-01-09 | p003 | 616 |
| 2020-08-18 | p001 | 1563 |
| 2020-09-12 | p002 | 730 |
| 2020-10-04 | p003 | 587 |
| 2021-02-15 | p001 | 1979 |
| 2021-02-22 | p002 | 134 |
| 2021-03-14 | p003 | 938 |

### Transformation:

You can use the following Window transformation to calculate the rolling minimum, maximum, and mode dates for the last five orders for each product identifier:

| Transformation Name | Window |
| --- | --- |
| Parameter: Formula1 | ROLLINGMINDATE(orderDate, 4, 0) |
| Parameter: Formula2 | ROLLINGMAXDATE(orderDate, 4, 0) |
| Parameter: Formula3 | ROLLINGMODEDATE(orderDate, 4, 0) |
| Parameter: Group by | prodId |
| Parameter: Order by | prodId |

You can use the following transformation to rename the generated `window` columns:

| Transformation Name | Rename columns |
| --- | --- |
| Parameter: Option | Manual rename |
| Parameter: Column | window1 |
| Parameter: New column name | rollingMinDate |
| Parameter: Parameter: Column | window2 |
| Parameter: New column name | rollingMaxDate |
| Parameter: Parameter: Column | window3 |
| Parameter: New column name | rollingModeDate |

**Results:**

| orderDate | prodId | orderDollars | rollingMinDate | rollingMaxDate | rollingModeDate |
| --- | --- | --- | --- | --- | --- |
| 3/16/20 | p003 | 1374 | 3/16/20 | 3/16/20 | 3/16/20 |
| 8/9/20 | p003 | 984 | 3/16/20 | 8/9/20 | 3/16/20 |
| 7/16/20 | p003 | 824 | 3/16/20 | 8/9/20 | 3/16/20 |
| 10/22/20 | p003 | 1513 | 3/16/20 | 10/22/20 | 3/16/20 |
| 3/23/20 | p003 | 1122 | 3/16/20 | 10/22/20 | 3/16/20 |
| 1/9/21 | p003 | 616 | 3/23/20 | 1/9/21 | 3/23/20 |
| 10/4/20 | p003 | 587 | 3/23/20 | 1/9/21 | 3/23/20 |
| 3/14/21 | p003 | 938 | 3/23/20 | 3/14/21 | 3/23/20 |
| 3/13/20 | p001 | 1445 | 3/13/20 | 3/13/20 | 3/13/20 |
| 3/23/20 | p001 | 1675 | 3/13/20 | 3/23/20 | 3/13/20 |
| 5/2/20 | p001 | 1395 | 3/13/20 | 5/2/20 | 3/13/20 |
| 9/2/20 | p001 | 1785 | 3/13/20 | 9/2/20 | 3/13/20 |
| 3/17/20 | p001 | 768 | 3/13/20 | 9/2/20 | 3/13/20 |
| 4/6/20 | p001 | 805 | 3/17/20 | 9/2/20 | 3/17/20 |
| 8/18/20 | p001 | 1563 | 3/17/20 | 9/2/20 | 3/17/20 |

| | | | | | |
|---|---|---|---|---|---|
| 2/15/21 | p001 | 1979 | 3/17/20 | 2/15/21 | 3/17/20 |
| 3/6/20 | p002 | 712 | 3/6/20 | 3/6/20 | 3/6/20 |
| 4/9/20 | p002 | 1005 | 3/6/20 | 4/9/20 | 3/6/20 |
| 6/14/20 | p002 | 1866 | 3/6/20 | 6/14/20 | 3/6/20 |
| 8/31/20 | p002 | 697 | 3/6/20 | 8/31/20 | 3/6/20 |
| 3/21/20 | p002 | 1893 | 3/6/20 | 8/31/20 | 3/6/20 |
| 5/9/20 | p002 | 1752 | 3/21/20 | 8/31/20 | 3/21/20 |
| 9/12/20 | p002 | 730 | 3/21/20 | 9/12/20 | 3/21/20 |
| 2/22/21 | p002 | 134 | 3/21/20 | 2/22/21 | 3/21/20 |

# ROLLINGMINDATE Function

**Contents:**

---

Computes the rolling minimum of Date values forward or backward of the current row within the specified column. Inputs must be of Datetime type.

- If an input value is missing or null, it is not factored in the computation. For example, for the first row in the dataset, the rolling minimum of previous values is undefined.
- The row from which to extract a value is determined by the order in which the rows are organized based on the `order` parameter.
- If you are working on a randomly generated sample of your dataset, the values that you see for this function might not correspond to the values that are generated on the full dataset during job execution.
- The function takes a column name and two optional integer parameters that determine the window backward and forward of the current row.
    - The default integer parameter values are `-1` and `0`, which computes the rolling function from the current row back to the first row of the dataset.
- This function works with the Window transform. See *Window Transform*.

For more information on a non-rolling version of this function, see *MINDATE Function*.

**Wrangle vs. SQL:** This function is part of Wrangle , a proprietary data transformation language. Wrangle is not SQL. For more information, see *Wrangle Language*.

## Basic Usage

**Column example:**

```
rollingmindate(myDates)
```

**Output:** Returns the rolling minimum of all Datetime values in the `myDates` column.

**Rows before example:**

```
rollingmindate(myDates, 3)
```

**Output:** Returns the rolling minimum of the current row and the three previous row values in the `myDates` column.

**Rows before and after example:**

```
rollingmindate(myDates, 3, 2)
```

**Output:** Returns the rolling minimum of the three previous row values, the current row value, and the two rows after the current one in the `myDates` column.

## Syntax and Arguments

```
rollingmindate(col_ref, rowsBefore_integer, rowsAfter_integer) order: order_col [group:
group_col]
```

| Argument | Required? | Data Type | Description |
|---|---|---|---|
| col_ref | Y | string | Name of column whose values are applied to the function |
| rowsBefore_integer | N | integer | Number of rows before the current one to include in the computation |
| rowsAfter_integer | N | integer | Number of rows after the current one to include in the computation |

For more information on the `order` and `group` parameters, see *Window Transform*.

For more information on syntax standards, see *Language Documentation Syntax Notes*.

**col_ref**

Name of the column whose values are used to compute the function. Inputs must be Datetime values.

Multiple columns and wildcards are not supported.

**Usage Notes:**

| Required? | Data Type | Example Value |
|---|---|---|
| Yes | String (column reference to Datetime values) | transactionDate |

**rowsBefore_integer, rowsAfter_integer**

Integers representing the number of rows before or after the current one from which to compute the rolling function, including the current row. For example, if the first value is 5, the current row and the five rows before it are used in the computation. Negative values for `rowsAfter_integer` compute the rolling function from rows preceding the current one.

- `rowBefore=0` generates the current row value only.
- `rowBefore=-1` uses all rows preceding the current one.
- If `rowsAfter` is not specified, then the value `0` is applied.
- If a `group` parameter is applied, then these parameter values should be no more than the maximum number of rows in the groups.

**Usage Notes:**

| Required? | Data Type | Example Value |
|---|---|---|
| No | Integer | 4 |

Examples

> **Tip:** For additional examples, see *Common Tasks*.

**Example - Rolling date functions**

This example describes how to use rolling functions for Datetime values.

**Functions**:

| Item | Description |
|---|---|
| ROLLINGMINDATE Function | Computes the rolling minimum of Date values forward or backward of the current row within the specified column. Inputs must be of Datetime type. |
| ROLLINGMAXDATE Function | Computes the rolling maximum of date values forward or backward of the current row within the specified column. Inputs must be of Datetime type. |
| ROLLINGMODEDATE Function | Computes the rolling mode (most common value) forward or backward of the current row within the specified column. Input values must be of Datetime data type. |

**Source:**

The following table contains an unordered list of orders:

| myDate | prodId | orderDollars |
|---|---|---|
| 2020-03-13 | p001 | 1445 |
| 2020-03-06 | p002 | 712 |
| 2020-03-16 | p003 | 1374 |
| 2020-03-23 | p001 | 1675 |
| 2020-04-09 | p002 | 1005 |
| 2020-08-09 | p003 | 984 |
| 2020-05-02 | p001 | 1395 |
| 2020-06-14 | p002 | 1866 |
| 2020-07-16 | p003 | 824 |
| 2020-09-02 | p001 | 1785 |
| 2020-08-31 | p002 | 697 |
| 2020-10-22 | p003 | 1513 |
| 2020-03-17 | p001 | 768 |
| 2020-03-21 | p002 | 1893 |
| 2020-03-23 | p003 | 1122 |
| 2020-04-06 | p001 | 805 |
| 2020-05-09 | p002 | 1752 |
| 2021-01-09 | p003 | 616 |
| 2020-08-18 | p001 | 1563 |
| 2020-09-12 | p002 | 730 |
| 2020-10-04 | p003 | 587 |
| 2021-02-15 | p001 | 1979 |
| 2021-02-22 | p002 | 134 |
| 2021-03-14 | p003 | 938 |

**Transformation:**

You can use the following Window transformation to calculate the rolling minimum, maximum, and mode dates for the last five orders for each product identifier:

| Transformation Name | Window |
|---|---|
| Parameter: Formula1 | ROLLINGMINDATE(orderDate, 4, 0) |
| Parameter: Formula2 | ROLLINGMAXDATE(orderDate, 4, 0) |
| Parameter: Formula3 | ROLLINGMODEDATE(orderDate, 4, 0) |
| Parameter: Group by | prodId |
| Parameter: Order by | prodId |

You can use the following transformation to rename the generated `window` columns:

| Transformation Name | Rename columns |
|---|---|
| Parameter: Option | Manual rename |
| Parameter: Column | window1 |
| Parameter: New column name | rollingMinDate |
| Parameter: Parameter: Column | window2 |
| Parameter: New column name | rollingMaxDate |
| Parameter: Parameter: Column | window3 |
| Parameter: New column name | rollingModeDate |

**Results:**

| orderDate | prodId | orderDollars | rollingMinDate | rollingMaxDate | rollingModeDate |
|---|---|---|---|---|---|
| 3/16/20 | p003 | 1374 | 3/16/20 | 3/16/20 | 3/16/20 |
| 8/9/20 | p003 | 984 | 3/16/20 | 8/9/20 | 3/16/20 |
| 7/16/20 | p003 | 824 | 3/16/20 | 8/9/20 | 3/16/20 |
| 10/22/20 | p003 | 1513 | 3/16/20 | 10/22/20 | 3/16/20 |
| 3/23/20 | p003 | 1122 | 3/16/20 | 10/22/20 | 3/16/20 |
| 1/9/21 | p003 | 616 | 3/23/20 | 1/9/21 | 3/23/20 |
| 10/4/20 | p003 | 587 | 3/23/20 | 1/9/21 | 3/23/20 |
| 3/14/21 | p003 | 938 | 3/23/20 | 3/14/21 | 3/23/20 |
| 3/13/20 | p001 | 1445 | 3/13/20 | 3/13/20 | 3/13/20 |
| 3/23/20 | p001 | 1675 | 3/13/20 | 3/23/20 | 3/13/20 |
| 5/2/20 | p001 | 1395 | 3/13/20 | 5/2/20 | 3/13/20 |
| 9/2/20 | p001 | 1785 | 3/13/20 | 9/2/20 | 3/13/20 |
| 3/17/20 | p001 | 768 | 3/13/20 | 9/2/20 | 3/13/20 |
| 4/6/20 | p001 | 805 | 3/17/20 | 9/2/20 | 3/17/20 |
| 8/18/20 | p001 | 1563 | 3/17/20 | 9/2/20 | 3/17/20 |

| | | | | | |
|---|---|---|---|---|---|
| 2/15/21 | p001 | 1979 | 3/17/20 | 2/15/21 | 3/17/20 |
| 3/6/20 | p002 | 712 | 3/6/20 | 3/6/20 | 3/6/20 |
| 4/9/20 | p002 | 1005 | 3/6/20 | 4/9/20 | 3/6/20 |
| 6/14/20 | p002 | 1866 | 3/6/20 | 6/14/20 | 3/6/20 |
| 8/31/20 | p002 | 697 | 3/6/20 | 8/31/20 | 3/6/20 |
| 3/21/20 | p002 | 1893 | 3/6/20 | 8/31/20 | 3/6/20 |
| 5/9/20 | p002 | 1752 | 3/21/20 | 8/31/20 | 3/21/20 |
| 9/12/20 | p002 | 730 | 3/21/20 | 9/12/20 | 3/21/20 |
| 2/22/21 | p002 | 134 | 3/21/20 | 2/22/21 | 3/21/20 |

# ROLLINGSUM Function

**Contents:**

Computes the rolling sum of values forward or backward of the current row within the specified column.

- If an input value is missing or null, it is not factored in the computation. For example, for the first row in the dataset, the rolling sum of previous values is the value in the first row.
- The row from which to extract a value is determined by the order in which the rows are organized based on the `order` parameter.
- If you are working on a randomly generated sample of your dataset, the values that you see for this function might not correspond to the values that are generated on the full dataset during job execution.
- The function takes a column name and two optional integer parameters that determine the window backward and forward of the current row.
    - The default integer parameter values are `-1` and `0`, which computes the rolling average from the current row back to the first row of the dataset.
- This function works with the Window transform. See *Window Transform*.

**Wrangle vs. SQL:** This function is part of Wrangle , a proprietary data transformation language. Wrangle is not SQL. For more information, see *Wrangle Language*.

## Basic Usage

**Column example:**

```
rollingsum(myCol)
```

**Output:** Returns the rolling sum of all values in the `myCol` column.

**Rows before example:**

```
rollingsum(myNumber, 3)
```

**Output:** Returns the rolling sum of the current row and the three previous row values in the `myNumber` column.

**Rows before and after example:**

```
rollingsum(myNumber, 3, 2)
```

**Output:** Returns the rolling sum of the three previous row values, the current row value, and the two rows after the current one in the `myNumber` column.

## Syntax and Arguments

```
rollingsum(col_ref, rowsBefore_integer, rowsAfter_integer) order: order_col [group:
group_col]
```

| Argument | Required? | Data Type | Description |
|---|---|---|---|
| col_ref | Y | string | Name of column whose values are applied to the function |
| rowsBefore_integer | N | integer | Number of rows before the current one to include in the computation. |
| rowsAfter_integer | N | integer | Number of rows after the current one to include in the computation |

For more information on the `order` and `group` parameters, see *Window Transform*.

For more information on syntax standards, see *Language Documentation Syntax Notes*.

**col_ref**

Name of the column whose values are used to compute the rolling sum.

- Multiple columns and wildcards are not supported.

**Usage Notes:**

| Required? | Data Type | Example Value |
|---|---|---|
| Yes | String (column reference to Integer or Decimal values) | `myColumn` |

**rowsBefore_integer, rowsAfter_integer**

Integers representing the number of rows before or after the current one from which to compute the rolling sum, including the current row. For example, if the first value is 5, the current row and the five rows before it are used in the computation. Negative values for `rowsAfter_integer` compute the rolling function from rows preceding the current one.

- `rowBefore=0` generates the current row value only.
- `rowBefore=-1` uses all rows preceding the current one.
- If `rowsAfter` is not specified, then the value `0` is applied.
- If a `group` parameter is applied, then these parameter values should be no more than the maximum number of rows in the groups.

**Usage Notes:**

| Required? | Data Type | Example Value |
|---|---|---|
| No | Integer | 4 |

## Examples

> **Tip:** For additional examples, see *Common Tasks*.

**Example - Rolling window functions**

This example describes how to use rolling computational functions.

**Functions**:

| Item | Description |
|---|---|
| ROLLINGSUM Function | Computes the rolling sum of values forward or backward of the current row within the specified column. |
| ROLLINGAVERAGE Function | Computes the rolling average of values forward or backward of the current row within the specified column. |
| ROWNUMBER Function | Generates a new column containing the row number as sorted by the `order` parameter and optionally grouped by the `group` parameter. |

Also:

| Item | Description |
|---|---|
| MONTH Function | Derives the month integer value from a Datetime value. Source value can be a a reference to a column containing Datetime values or a literal. |
| FLOOR Function | Computes the largest integer that is not more than the input value. Input can be an Integer, a Decimal, a column reference, or an expression. |
| ROWNUMBER Function | Generates a new column containing the row number as sorted by the `order` parameter and optionally grouped by the `group` parameter. |

The following dataset contains sales data over the final quarter of the year.

**Source:**

| Date | Sales |
|---|---|
| 10/2/16 | 200 |
| 10/9/16 | 500 |
| 10/16/16 | 350 |
| 10/23/16 | 400 |
| 10/30/16 | 190 |
| 11/6/16 | 550 |
| 11/13/16 | 610 |
| 11/20/16 | 480 |
| 11/27/16 | 660 |
| 12/4/16 | 690 |
| 12/11/16 | 810 |
| 12/18/16 | 950 |
| 12/25/16 | 1020 |
| 1/1/17 | 680 |

**Transformation:**

First, you want to maintain the row information as a separate column. Since data is ordered already by the `Date` column, you can use the following:

| | |
|---|---|
| **Transformation Name** | `Window` |
| **Parameter: Formulas** | `ROWNUMBER()` |
| | |

| | |
|---|---|
| **Parameter: Order by** | Date |

Rename this column to `rowId` for week of quarter.

Now, you want to extract month and week information from the `Date` values. Deriving the month value:

| | |
|---|---|
| **Transformation Name** | New formula |
| **Parameter: Formula type** | Single row formula |
| **Parameter: Formula** | MONTH(Date) |
| **Parameter: New column name** | 'Month' |

Deriving the quarter value:

| | |
|---|---|
| **Transformation Name** | New formula |
| **Parameter: Formula type** | Single row formula |
| **Parameter: Formula** | (1 + FLOOR(((month-1)/3))) |
| **Parameter: New column name** | 'QTR' |

Deriving the week-of-quarter value:

| | |
|---|---|
| **Transformation Name** | Window |
| **Parameter: Formulas** | ROWNUMBER() |
| **Parameter: Group by** | QTR |
| **Parameter: Order by** | Date |

Rename this column `WOQ` (week of quarter).

Deriving the week-of-month value:

| | |
|---|---|
| **Transformation Name** | Window |
| **Parameter: Formulas** | ROWNUMBER() |
| **Parameter: Group by** | Month |
| **Parameter: Order by** | Date |

Rename this column `WOM` (week of month).

Now, you perform your rolling computations. Compute the running total of sales using the following:

| | |
|---|---|
| **Transformation Name** | Window |
| **Parameter: Formulas** | ROLLINGSUM(Sales, -1, 0) |
| **Parameter: Group by** | QTR |
| **Parameter: Order by** | Date |

The `-1` parameter is used in the above computation to gather the rolling sum of all rows of data from the current one to the first one. Note that the use of the `QTR` column for grouping, which moves the value for the `01/01/2017` into its own computational bucket. This may or may not be preferred.

Rename this column `QTD` (quarter to-date). Now, generate a similar column to compute the rolling average of weekly sales for the quarter:

| Transformation Name | Window |
|---|---|
| Parameter: Formulas | ROUND(ROLLINGAVERAGE(Sales, -1, 0)) |
| Parameter: Group by | QTR |
| Parameter: Order by | Date |

Since the `ROLLINGAVERAGE` function can compute fractional values, it is wrapped in the `ROUND` function for neatness. Rename this column `avgWeekByQuarter`.

**Results:**

When the unnecessary columns are dropped and some reordering is applied, your dataset should look like the following:

| Date | WOQ | Sales | QTD | avgWeekByQuarter |
|---|---|---|---|---|
| 10/2/16 | 1 | 200 | 200 | 200 |
| 10/9/16 | 2 | 500 | 700 | 350 |
| 10/16/16 | 3 | 350 | 1050 | 350 |
| 10/23/16 | 4 | 400 | 1450 | 363 |
| 10/30/16 | 5 | 190 | 1640 | 328 |
| 11/6/16 | 6 | 550 | 2190 | 365 |
| 11/13/16 | 7 | 610 | 2800 | 400 |
| 11/20/16 | 8 | 480 | 3280 | 410 |
| 11/27/16 | 9 | 660 | 3940 | 438 |
| 12/4/16 | 10 | 690 | 4630 | 463 |
| 12/11/16 | 11 | 810 | 5440 | 495 |
| 12/18/16 | 12 | 950 | 6390 | 533 |
| 12/25/16 | 13 | 1020 | 7410 | 570 |
| 1/1/17 | 1 | 680 | 680 | 680 |

# ROLLINGSTDEV Function

**Contents:**

- *Basic Usage*
- *Syntax and Arguments*
  - *col_ref*
  - *rowsBefore_integer, rowsAfter_integer*
- *Examples*
  - *Example - Rolling computations for racing splits*

---

Computes the rolling standard deviation of values forward or backward of the current row within the specified column.

- If an input value is missing or null, it is not factored in the computation. For example, for the first row in the dataset, the rolling standard deviation of previous values is undefined.
- The row from which to extract a value is determined by the order in which the rows are organized based on the `order` parameter.
- If you are working on a randomly generated sample of your dataset, the values that you see for this function might not correspond to the values that are generated on the full dataset during job execution.
- The function takes a column name and two optional integer parameters that determine the window backward and forward of the current row.
  - The default integer parameter values are `-1` and `0`, which computes the rolling function from the current row back to the first row of the dataset.
- This function works with the Window transform. See *Window Transform*.

**Relevant terms:**

| Term | Description |
|------|-------------|
| Population | Population statistical functions are computed from all possible values. See *https://en.wikipedia.org/wiki/Statistical_population*. |
| Sample | Sample-based statistical functions are computed from a subset or sample of all values. See *https://en.wikipedia.org/wiki/Sampling_(statistics)*.<br><br>These function names include $SAMP$ in their name.<br><br>**NOTE:** Statistical sampling has no relationship to the samples taken within the product. When statistical functions are computed during job execution, they are applied across the entire dataset. Sample method calculations are computed at that time. |

For more information on a non-rolling version of this function, see *STDEV Function*.

**Wrangle vs. SQL:** This function is part of Wrangle , a proprietary data transformation language. Wrangle is not SQL. For more information, see *Wrangle Language*.

## Basic Usage

**Column example:**

```
rollingstdev(myCol)
```

**Output:** Returns the rolling standard deviation of all values in the `myCol` column.

**Rows before example:**

```
rollingstdev(myNumber, 100)
```

**Output:** Returns the rolling standard deviation of the current row and the 100 previous row values in the `myNumber` column.

**Rows before and after example:**

```
rollingstdev(myNumber, 3, 2)
```

**Output:** Returns the rolling standard deviation of the three previous row values, the current row value, and the two rows after the current one in the `myNumber` column.

## Syntax and Arguments

```
rollingstdev(col_ref, rowsBefore_integer, rowsAfter_integer) order: order_col [group:
group_col]
```

| Argument | Required? | Data Type | Description |
|---|---|---|---|
| col_ref | Y | string | Name of column whose values are applied to the function |
| rowsBefore_integer | N | integer | Number of rows before the current one to include in the computation |
| rowsAfter_integer | N | integer | Number of rows after the current one to include in the computation |

For more information on the `order` and `group` parameters, see *Window Transform*.

For more information on syntax standards, see *Language Documentation Syntax Notes*.

**col_ref**

Name of the column whose values are used to compute the function.

- Multiple columns and wildcards are not supported.

**Usage Notes:**

| Required? | Data Type | Example Value |
|---|---|---|
| Yes | String (column reference to Integer or Decimal values) | `myColumn` |

**rowsBefore_integer, rowsAfter_integer**

Integers representing the number of rows before or after the current one from which to compute the rolling function, including the current row. For example, if the first value is `5`, the current row and the five before after it are used in the computation. Negative values for `rowsAfter_integer` compute the rolling function from rows preceding the current one.

- `rowBefore=0` generates the current row value only.
- `rowBefore=-1` uses all rows preceding the current one.
- If `rowsAfter` is not specified, then the value `0` is applied.

- If a `group` parameter is applied, then these parameter values should be no more than the maximum number of rows in the groups.

**Usage Notes:**

| Required? | Data Type | Example Value |
|---|---|---|
| No | Integer | 4 |

## Examples

> **Tip:** For additional examples, see *Common Tasks*.

**Example - Rolling computations for racing splits**

This example describes how to use rolling statistical functions.

**Functions**:

| Item | Description |
|---|---|
| ROLLINGAVERAGE Function | Computes the rolling average of values forward or backward of the current row within the specified column. |
| ROLLINGMAX Function | Computes the rolling maximum of values forward or backward of the current row within the specified column. Inputs can be Integer, Decimal, or Datetime. |
| ROLLINGSTDEV Function | Computes the rolling standard deviation of values forward or backward of the current row within the specified column. |
| ROLLINGVAR Function | Computes the rolling variance of values forward or backward of the current row within the specified column. |
| ROLLINGSTDEVSAMP Function | Computes the rolling standard deviation of values forward or backward of the current row within the specified column using the sample statistical method. |
| ROLLINGVARSAMP Function | Computes the rolling variance of values forward or backward of the current row within the specified column using the sample statistical method. |

Also:

| Item | Description |
|---|---|
| MERGE Function | Merges two or more columns of String type to generate output of String type. Optionally, you can insert a delimiter between the merged values. |
| ROUND Function | Rounds input value to the nearest integer. Input can be an Integer, a Decimal, a column reference, or an expression. Optional second argument can be used to specify the number of digits to which to round. |

**Source:**

In this example, the following data comes from times recorded at regular intervals during a three-lap race around a track. The source data is in cumulative time in seconds (`time_sc`). You can use ROLLING and other windowing functions to break down the data into more meaningful metrics.

| lap | quarter | time_sc |
|---|---|---|
| 1 | 0 | 0.000 |
| 1 | 1 | 19.554 |
| 1 | 2 | 39.785 |

| | | |
|---|---|---|
| 1 | 3 | 60.021 |
| 2 | 0 | 80.950 |
| 2 | 1 | 101.785 |
| 2 | 2 | 121.005 |
| 2 | 3 | 141.185 |
| 3 | 0 | 162.008 |
| 3 | 1 | 181.887 |
| 3 | 2 | 200.945 |
| 3 | 3 | 220.856 |

**Transformation:**

**Primary key:** Since the quarter information repeats every lap, there is no unique identifier for each row. The following steps create this identifier:

| | |
|---|---|
| Transformation Name | `Change column data type` |
| Parameter: Columns | `lap,quarter` |
| Parameter: New type | `String` |

| | |
|---|---|
| Transformation Name | `New formula` |
| Parameter: Formula type | `Single row formula` |
| Parameter: Formula | `MERGE(['l',lap,'q',quarter])` |
| Parameter: New column name | `'splitId'` |

**Get split times:** Use the following transform to break down the splits for each quarter of the race:

| | |
|---|---|
| Transformation Name | `New formula` |
| Parameter: Formula type | `Multiple row formula` |
| Parameter: Formula | `ROUND(time_sc - PREV(time_sc, 1), 3)` |
| Parameter: Order rows by | `splitId` |
| Parameter: New column name | `'split_time_sc'` |

**Compute rolling computations:** You can use the following types of computations to provide rolling metrics on the current and three previous splits:

| | |
|---|---|
| Transformation Name | `New formula` |
| Parameter: Formula type | `Multiple row formula` |
| Parameter: Formula | `ROLLINGAVERAGE(split_time_sc, 3)` |
| Parameter: Order rows by | `splitId` |
| Parameter: New column | `'ravg'` |

| | |
|---|---|
| **name** | |

| | |
|---|---|
| **Transformation Name** | `New formula` |
| **Parameter: Formula type** | `Multiple row formula` |
| **Parameter: Formula** | `ROLLINGMAX(split_time_sc, 3)` |
| **Parameter: Order rows by** | `splitId` |
| **Parameter: New column name** | `'rmax'` |

| | |
|---|---|
| **Transformation Name** | `New formula` |
| **Parameter: Formula type** | `Multiple row formula` |
| **Parameter: Formula** | `ROLLINGMIN(split_time_sc, 3)` |
| **Parameter: Order rows by** | `splitId` |
| **Parameter: New column name** | `'rmin'` |

| | |
|---|---|
| **Transformation Name** | `New formula` |
| **Parameter: Formula type** | `Multiple row formula` |
| **Parameter: Formula** | `ROUND(ROLLINGSTDEV(split_time_sc, 3), 3)` |
| **Parameter: Order rows by** | `splitId` |
| **Parameter: New column name** | `'rstdev'` |

| | |
|---|---|
| **Transformation Name** | `New formula` |
| **Parameter: Formula type** | `Multiple row formula` |
| **Parameter: Formula** | `ROUND(ROLLINGVAR(split_time_sc, 3), 3)` |
| **Parameter: Order rows by** | `splitId` |
| **Parameter: New column name** | `'rvar'` |

**Compute rolling computations using sample method:** These metrics compute the rolling STDEV and VAR on the current and three previous splits using the sample method:

| | |
|---|---|
| **Transformation Name** | `New formula` |
| **Parameter: Formula type** | `Multiple row formula` |
| **Parameter: Formula** | `ROUND(ROLLINGSTDEVSAMP(split_time_sc, 3), 3)` |
| **Parameter: Order rows by** | `splitId` |
| **Parameter: New column name** | `'rstdev_samp'` |

| | |
|---|---|
| | |

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | `Multiple row formula` |
| Parameter: Formula | `ROUND(ROLLINGVARSAMP(split_time_sc, 3), 3)` |
| Parameter: Order rows by | `splitId` |
| Parameter: New column name | `'rvar_samp'` |

### Results:

When the above transforms have been completed, the results look like the following:

| lap | quarter | splitId | time_sc | split_time_sc | rvar_samp | rstdev_samp | rvar | rstdev | rmin | rmax | ravg |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 0 | l1q0 | 0 | | | | | | | | |
| 1 | 1 | l1q1 | 20.096 | 20.096 | | | 0 | 0 | 20.096 | 20.096 | 20.096 |
| 1 | 2 | l1q2 | 40.53 | 20.434 | 0.229 | 0.479 | 0.029 | 0.169 | 20.096 | 20.434 | 20.265 |
| 1 | 3 | l1q3 | 61.031 | 20.501 | 0.154 | 0.392 | 0.031 | 0.177 | 20.096 | 20.501 | 20.344 |
| 2 | 0 | l2q0 | 81.087 | 20.056 | 0.315 | 0.561 | 0.039 | 0.198 | 20.056 | 20.501 | 20.272 |
| 2 | 1 | l2q1 | 101.383 | 20.296 | 0.142 | 0.376 | 0.029 | 0.17 | 20.056 | 20.501 | 20.322 |
| 2 | 2 | l2q2 | 122.092 | 20.709 | 0.617 | 0.786 | 0.059 | 0.242 | 20.056 | 20.709 | 20.39 |
| 2 | 3 | l2q3 | 141.886 | 19.794 | 0.621 | 0.788 | 0.113 | 0.337 | 19.794 | 20.709 | 20.214 |
| 3 | 0 | l3q0 | 162.581 | 20.695 | 0.579 | 0.761 | 0.139 | 0.373 | 19.794 | 20.709 | 20.373 |
| 3 | 1 | l3q1 | 183.018 | 20.437 | 0.443 | 0.666 | 0.138 | 0.371 | 19.794 | 20.709 | 20.409 |
| 3 | 2 | l3q2 | 203.493 | 20.475 | 0.537 | 0.733 | 0.113 | 0.336 | 19.794 | 20.695 | 20.35 |
| 3 | 3 | l3q3 | 222.893 | 19.4 | 0.520 | 0.721 | 0.252 | 0.502 | 19.4 | 20.695 | 20.252 |

You can reduce the number of steps by applying a `window` transform such as the following:

| Transformation Name | `Window` |
|---|---|
| Parameter: Formula1 | `lap` |
| Parameter: Formula2 | `rollingaverage(split_time_sc, 0, 3)` |
| Parameter: Formula3 | `rollingmax(split_time_sc, 0, 3)` |
| Parameter: Formula4 | `rollingmin(split_time_sc, 0, 3)` |
| Parameter: Formula5 | `round(rollingstdev(split_time_sc, 0, 3), 3)` |
| Parameter: Formula6 | `round(rollingvar(split_time_sc, 0, 3), 3)` |
| Parameter: Formula7 | `round(rollingstdevsamp(split_time_sc, 0, 3), 3)` |
| Parameter: Formula8 | `round(rollingvarsamp(split_time_sc, 0, 3), 3)` |
| Parameter: Group by | `lap` |
| Parameter: Order by | `lap` |

However, you must rename all of the generated `windowX` columns.

# ROLLINGSTDEVSAMP Function

**Contents:**

Computes the rolling standard deviation of values forward or backward of the current row within the specified column using the sample statistical method.

- If an input value is missing or null, it is not factored in the computation. For example, for the first row in the dataset, the rolling standard deviation of previous values is undefined.
- The row from which to extract a value is determined by the order in which the rows are organized based on the `order` parameter.
- If you are working on a randomly generated sample of your dataset, the values that you see for this function might not correspond to the values that are generated on the full dataset during job execution.
- The function takes a column name and two optional integer parameters that determine the window backward and forward of the current row.
    - The default integer parameter values are `-1` and `0`, which computes the rolling function from the current row back to the first row of the dataset.
- This function works with the Window transform. See *Window Transform*.

> **NOTE:** This function applies to a sample of the entire population. More information is below.

**Relevant terms:**

| Term | Description |
|---|---|
| Population | Population statistical functions are computed from all possible values. See *https://en.wikipedia.org/wiki/Statistical_population*. |
| Sample | Sample-based statistical functions are computed from a subset or sample of all values. See *https://en.wikipedia.org/wiki/Sampling_(statistics)*. <br><br> These function names include $SAMP$ in their name. <br><br> > **NOTE:** Statistical sampling has no relationship to the samples taken within the product. When statistical functions are computed during job execution, they are applied across the entire dataset. Sample method calculations are computed at that time. |

For more information on a non-rolling version of this function, see *STDEV Function*.

**Wrangle vs. SQL:** This function is part of Wrangle , a proprietary data transformation language. Wrangle is not SQL. For more information, see *Wrangle Language*.

## Basic Usage

**Column example:**

```
rollingstdevsamp(myCol)
```

**Output:** Returns the rolling standard deviation of all values in the `myCol` column using the sample method of calculation.

**Rows before example:**

```
<span>rollingstdevsamp</span>(myNumber, 100)
```

**Output:** Returns the rolling standard deviation of the current row and the 100 previous row values in the `myNumber` column using the sample method of calculation.

**Rows before and after example:**

```
<span>rollingstdevsamp</span>(myNumber, 3, 2)
```

**Output:** Returns the rolling standard deviation of the three previous row values, the current row value, and the two rows after the current one in the `myNumber` column.

## Syntax and Arguments

```
<span>rollingstdevsamp</span>(col_ref, rowsBefore_integer, rowsAfter_integer) order:
order_col [group: group_col]
```

| Argument | Required? | Data Type | Description |
|---|---|---|---|
| col_ref | Y | string | Name of column whose values are applied to the function |
| rowsBefore_integer | N | integer | Number of rows before the current one to include in the computation |
| rowsAfter_integer | N | integer | Number of rows after the current one to include in the computation |

For more information on the `order` and `group` parameters, see *Window Transform*.

For more information on syntax standards, see *Language Documentation Syntax Notes*.

**col_ref**

Name of the column whose values you wish to use in the calculation. Column must be a numeric (Integer or Decimal) type.

- Multiple columns and wildcards are not supported.

**Usage Notes:**

| Required? | Data Type | Example Value |
|---|---|---|
| Yes | String (column reference to Integer or Decimal values) | `myColumn` |

**rowsBefore_integer, rowsAfter_integer**

Integers representing the number of rows before or after the current one from which to compute the rolling function, including the current row. For example, if the first value is 5, the current row and the five before after it are used in the computation. Negative values for `rowsAfter_integer` compute the rolling function from rows preceding the current one.

- `rowBefore=0` generates the current row value only.
- `rowBefore=-1` uses all rows preceding the current one.
- If `rowsAfter` is not specified, then the value `0` is applied.
- If a `group` parameter is applied, then these parameter values should be no more than the maximum number of rows in the groups.

**Usage Notes:**

| Required? | Data Type | Example Value |
|---|---|---|
| No | Integer | 4 |

## Examples

> **Tip:** For additional examples, see *Common Tasks*.

**Example - Rolling computations for racing splits**

This example describes how to use rolling statistical functions.

**Functions**:

| Item | Description |
|---|---|
| ROLLINGAVERAGE Function | Computes the rolling average of values forward or backward of the current row within the specified column. |
| ROLLINGMAX Function | Computes the rolling maximum of values forward or backward of the current row within the specified column. Inputs can be Integer, Decimal, or Datetime. |
| ROLLINGSTDEV Function | Computes the rolling standard deviation of values forward or backward of the current row within the specified column. |
| ROLLINGVAR Function | Computes the rolling variance of values forward or backward of the current row within the specified column. |
| ROLLINGSTDEVSAMP Function | Computes the rolling standard deviation of values forward or backward of the current row within the specified column using the sample statistical method. |
| ROLLINGVARSAMP Function | Computes the rolling variance of values forward or backward of the current row within the specified column using the sample statistical method. |

Also:

| Item | Description |
|---|---|
| MERGE Function | Merges two or more columns of String type to generate output of String type. Optionally, you can insert a delimiter between the merged values. |
| ROUND Function | Rounds input value to the nearest integer. Input can be an Integer, a Decimal, a column reference, or an expression. Optional second argument can be used to specify the number of digits to which to round. |

**Source:**

In this example, the following data comes from times recorded at regular intervals during a three-lap race around a track. The source data is in cumulative time in seconds (`time_sc`). You can use ROLLING and other windowing functions to break down the data into more meaningful metrics.

| lap | quarter | time_sc |
|---|---|---|
| 1 | 0 | 0.000 |

| | | |
|---|---|---|
| 1 | 1 | 19.554 |
| 1 | 2 | 39.785 |
| 1 | 3 | 60.021 |
| 2 | 0 | 80.950 |
| 2 | 1 | 101.785 |
| 2 | 2 | 121.005 |
| 2 | 3 | 141.185 |
| 3 | 0 | 162.008 |
| 3 | 1 | 181.887 |
| 3 | 2 | 200.945 |
| 3 | 3 | 220.856 |

**Transformation:**

**Primary key:** Since the quarter information repeats every lap, there is no unique identifier for each row. The following steps create this identifier:

| | |
|---|---|
| **Transformation Name** | Change column data type |
| **Parameter: Columns** | lap,quarter |
| **Parameter: New type** | String |

| | |
|---|---|
| **Transformation Name** | New formula |
| **Parameter: Formula type** | Single row formula |
| **Parameter: Formula** | MERGE(['l',lap,'q',quarter]) |
| **Parameter: New column name** | 'splitId' |

**Get split times:** Use the following transform to break down the splits for each quarter of the race:

| | |
|---|---|
| **Transformation Name** | New formula |
| **Parameter: Formula type** | Multiple row formula |
| **Parameter: Formula** | ROUND(time_sc - PREV(time_sc, 1), 3) |
| **Parameter: Order rows by** | splitId |
| **Parameter: New column name** | 'split_time_sc' |

**Compute rolling computations:** You can use the following types of computations to provide rolling metrics on the current and three previous splits:

| | |
|---|---|
| **Transformation Name** | New formula |
| **Parameter: Formula type** | Multiple row formula |
| **Parameter: Formula** | ROLLINGAVERAGE(split_time_sc, 3) |

| Parameter: Order rows by | splitId |
|---|---|
| Parameter: New column name | 'ravg' |

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Multiple row formula |
| Parameter: Formula | ROLLINGMAX(split_time_sc, 3) |
| Parameter: Order rows by | splitId |
| Parameter: New column name | 'rmax' |

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Multiple row formula |
| Parameter: Formula | ROLLINGMIN(split_time_sc, 3) |
| Parameter: Order rows by | splitId |
| Parameter: New column name | 'rmin' |

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Multiple row formula |
| Parameter: Formula | ROUND(ROLLINGSTDEV(split_time_sc, 3), 3) |
| Parameter: Order rows by | splitId |
| Parameter: New column name | 'rstdev' |

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Multiple row formula |
| Parameter: Formula | ROUND(ROLLINGVAR(split_time_sc, 3), 3) |
| Parameter: Order rows by | splitId |
| Parameter: New column name | 'rvar' |

**Compute rolling computations using sample method:** These metrics compute the rolling STDEV and VAR on the current and three previous splits using the sample method:

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Multiple row formula |
| Parameter: Formula | ROUND(ROLLINGSTDEVSAMP(split_time_sc, 3), 3) |
| Parameter: Order rows by | splitId |
| Parameter: New column | 'rstdev_samp' |

| name |  |
|------|--|

| Transformation Name | New formula |
|---------------------|-------------|
| Parameter: Formula type | Multiple row formula |
| Parameter: Formula | ROUND(ROLLINGVARSAMP(split_time_sc, 3), 3) |
| Parameter: Order rows by | splitId |
| Parameter: New column name | 'rvar_samp' |

**Results:**

When the above transforms have been completed, the results look like the following:

| lap | quarter | splitId | time_sc | split_time_sc | rvar_samp | rstdev_samp | rvar | rstdev | rmin | rmax | ravg |
|-----|---------|---------|---------|---------------|-----------|-------------|------|--------|------|------|------|
| 1 | 0 | l1q0 | 0 | | | | | | | | |
| 1 | 1 | l1q1 | 20.096 | 20.096 | | | 0 | 0 | 20.096 | 20.096 | 20.096 |
| 1 | 2 | l1q2 | 40.53 | 20.434 | 0.229 | 0.479 | 0.029 | 0.169 | 20.096 | 20.434 | 20.265 |
| 1 | 3 | l1q3 | 61.031 | 20.501 | 0.154 | 0.392 | 0.031 | 0.177 | 20.096 | 20.501 | 20.344 |
| 2 | 0 | l2q0 | 81.087 | 20.056 | 0.315 | 0.561 | 0.039 | 0.198 | 20.056 | 20.501 | 20.272 |
| 2 | 1 | l2q1 | 101.383 | 20.296 | 0.142 | 0.376 | 0.029 | 0.17 | 20.056 | 20.501 | 20.322 |
| 2 | 2 | l2q2 | 122.092 | 20.709 | 0.617 | 0.786 | 0.059 | 0.242 | 20.056 | 20.709 | 20.39 |
| 2 | 3 | l2q3 | 141.886 | 19.794 | 0.621 | 0.788 | 0.113 | 0.337 | 19.794 | 20.709 | 20.214 |
| 3 | 0 | l3q0 | 162.581 | 20.695 | 0.579 | 0.761 | 0.139 | 0.373 | 19.794 | 20.709 | 20.373 |
| 3 | 1 | l3q1 | 183.018 | 20.437 | 0.443 | 0.666 | 0.138 | 0.371 | 19.794 | 20.709 | 20.409 |
| 3 | 2 | l3q2 | 203.493 | 20.475 | 0.537 | 0.733 | 0.113 | 0.336 | 19.794 | 20.695 | 20.35 |
| 3 | 3 | l3q3 | 222.893 | 19.4 | 0.520 | 0.721 | 0.252 | 0.502 | 19.4 | 20.695 | 20.252 |

You can reduce the number of steps by applying a `window` transform such as the following:

| Transformation Name | Window |
|---------------------|--------|
| Parameter: Formula1 | lap |
| Parameter: Formula2 | rollingaverage(split_time_sc, 0, 3) |
| Parameter: Formula3 | rollingmax(split_time_sc, 0, 3) |
| Parameter: Formula4 | rollingmin(split_time_sc, 0, 3) |
| Parameter: Formula5 | round(rollingstdev(split_time_sc, 0, 3), 3) |
| Parameter: Formula6 | round(rollingvar(split_time_sc, 0, 3), 3) |
| Parameter: Formula7 | round(rollingstdevsamp(split_time_sc, 0, 3), 3) |
| Parameter: Formula8 | round(rollingvarsamp(split_time_sc, 0, 3), 3) |
| Parameter: Group by | lap |
| Parameter: Order by | lap |

However, you must rename all of the generated `windowX` columns.

# ROLLINGVAR Function

**Contents:**

Computes the rolling variance of values forward or backward of the current row within the specified column.

- If an input value is missing or null, it is not factored in the computation. For example, for the first row in the dataset, the rolling variance of previous values is undefined.
- The row from which to extract a value is determined by the order in which the rows are organized based on the `order` parameter.
- If you are working on a randomly generated sample of your dataset, the values that you see for this function might not correspond to the values that are generated on the full dataset during job execution.
- The function takes a column name and two optional integer parameters that determine the window backward and forward of the current row.
    - The default integer parameter values are `-1` and `0`, which computes the rolling function from the current row back to the first row of the dataset.
- This function works with the Window transform. See *Window Transform*.

**Relevant terms:**

| Term | Description |
|---|---|
| Population | Population statistical functions are computed from all possible values. See *https://en.wikipedia.org/wiki/Statistical_population*. |
| Sample | Sample-based statistical functions are computed from a subset or sample of all values. See *https://en.wikipedia.org/wiki/Sampling_(statistics)*.<br><br>These function names include `SAMP` in their name.<br><br>**NOTE:** Statistical sampling has no relationship to the samples taken within the product. When statistical functions are computed during job execution, they are applied across the entire dataset. Sample method calculations are computed at that time. |

For more information on a non-rolling version of this function, see *VAR Function*.

**Wrangle vs. SQL:** This function is part of Wrangle , a proprietary data transformation language. Wrangle is not SQL. For more information, see *Wrangle Language*.

## Basic Usage

**Column example:**

```
rollingvar(myCol)
```

**Output:** Returns the rolling variance of all values in the `myCol` column from the first row of the dataset to the current one.

**Rows before example:**

```
rollingvar(myNumber, 100)
```

**Output:** Returns the rolling variance of the current row and the 100 previous row values in the `myNumber` column.

**Rows before and after example:**

```
rollingvar(myNumber, 3, 2)
```

**Output:** Returns the rolling variance of the three previous row values, the current row value, and the two rows after the current one in the `myNumber` column.

## Syntax and Arguments

```
rollingvar(col_ref, rowsBefore_integer, rowsAfter_integer) order: order_col [group:
group_col]
```

| Argument | Required? | Data Type | Description |
|---|---|---|---|
| col_ref | Y | string | Name of column whose values are applied to the function |
| rowsBefore_integer | N | integer | Number of rows before the current one to include in the computation |
| rowsAfter_integer | N | integer | Number of rows after the current one to include in the computation |

For more information on the `order` and `group` parameters, see *Window Transform*.

For more information on syntax standards, see *Language Documentation Syntax Notes*.

**col_ref**

Name of the column whose values are used to compute the function.

- Multiple columns and wildcards are not supported.

**Usage Notes:**

| Required? | Data Type | Example Value |
|---|---|---|
| Yes | String (column reference to Integer or Decimal values) | `myColumn` |

**rowsBefore_integer, rowsAfter_integer**

Integers representing the number of rows before or after the current one from which to compute the rolling function, including the current row. For example, if the first value is 5, the current row and the five rows before it are used in the computation. Negative values for `rowsAfter_integer` compute the rolling function from rows preceding the current one.

- `rowBefore=0` generates the current row value only.
- `rowBefore=-1` uses all rows preceding the current one.
- If `rowsAfter` is not specified, then the value 0 is applied.
- If a `group` parameter is applied, then these parameter values should be no more than the maximum number of rows in the groups.

**Usage Notes:**

| Required? | Data Type | Example Value |
|-----------|-----------|---------------|
| No | Integer | 4 |

## Examples

> **Tip:** For additional examples, see *Common Tasks*.

### Example - Rolling computations for racing splits

This example describes how to use rolling statistical functions.

**Functions**:

| Item | Description |
|------|-------------|
| ROLLINGAVERAGE Function | Computes the rolling average of values forward or backward of the current row within the specified column. |
| ROLLINGMAX Function | Computes the rolling maximum of values forward or backward of the current row within the specified column. Inputs can be Integer, Decimal, or Datetime. |
| ROLLINGSTDEV Function | Computes the rolling standard deviation of values forward or backward of the current row within the specified column. |
| ROLLINGVAR Function | Computes the rolling variance of values forward or backward of the current row within the specified column. |
| ROLLINGSTDEVSAMP Function | Computes the rolling standard deviation of values forward or backward of the current row within the specified column using the sample statistical method. |
| ROLLINGVARSAMP Function | Computes the rolling variance of values forward or backward of the current row within the specified column using the sample statistical method. |

Also:

| Item | Description |
|------|-------------|
| MERGE Function | Merges two or more columns of String type to generate output of String type. Optionally, you can insert a delimiter between the merged values. |
| ROUND Function | Rounds input value to the nearest integer. Input can be an Integer, a Decimal, a column reference, or an expression. Optional second argument can be used to specify the number of digits to which to round. |

**Source:**

In this example, the following data comes from times recorded at regular intervals during a three-lap race around a track. The source data is in cumulative time in seconds (`time_sc`). You can use ROLLING and other windowing functions to break down the data into more meaningful metrics.

| lap | quarter | time_sc |
|-----|---------|---------|
| 1 | 0 | 0.000 |
| 1 | 1 | 19.554 |
| 1 | 2 | 39.785 |
| 1 | 3 | 60.021 |
| 2 | 0 | 80.950 |
| 2 | 1 | 101.785 |

| | | |
|---|---|---|
| 2 | 2 | 121.005 |
| 2 | 3 | 141.185 |
| 3 | 0 | 162.008 |
| 3 | 1 | 181.887 |
| 3 | 2 | 200.945 |
| 3 | 3 | 220.856 |

**Transformation:**

**Primary key:** Since the quarter information repeats every lap, there is no unique identifier for each row. The following steps create this identifier:

| | |
|---|---|
| **Transformation Name** | `Change column data type` |
| **Parameter: Columns** | `lap,quarter` |
| **Parameter: New type** | `String` |

| | |
|---|---|
| **Transformation Name** | `New formula` |
| **Parameter: Formula type** | `Single row formula` |
| **Parameter: Formula** | `MERGE(['l',lap,'q',quarter])` |
| **Parameter: New column name** | `'splitId'` |

**Get split times:** Use the following transform to break down the splits for each quarter of the race:

| | |
|---|---|
| **Transformation Name** | `New formula` |
| **Parameter: Formula type** | `Multiple row formula` |
| **Parameter: Formula** | `ROUND(time_sc - PREV(time_sc, 1), 3)` |
| **Parameter: Order rows by** | `splitId` |
| **Parameter: New column name** | `'split_time_sc'` |

**Compute rolling computations:** You can use the following types of computations to provide rolling metrics on the current and three previous splits:

| | |
|---|---|
| **Transformation Name** | `New formula` |
| **Parameter: Formula type** | `Multiple row formula` |
| **Parameter: Formula** | `ROLLINGAVERAGE(split_time_sc, 3)` |
| **Parameter: Order rows by** | `splitId` |
| **Parameter: New column name** | `'ravg'` |

| | |
|---|---|
| **Transformation Name** | `New formula` |
| | |

| Parameter: Formula type | Multiple row formula |
|---|---|
| Parameter: Formula | ROLLINGMAX(split_time_sc, 3) |
| Parameter: Order rows by | splitId |
| Parameter: New column name | 'rmax' |

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Multiple row formula |
| Parameter: Formula | ROLLINGMIN(split_time_sc, 3) |
| Parameter: Order rows by | splitId |
| Parameter: New column name | 'rmin' |

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Multiple row formula |
| Parameter: Formula | ROUND(ROLLINGSTDEV(split_time_sc, 3), 3) |
| Parameter: Order rows by | splitId |
| Parameter: New column name | 'rstdev' |

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Multiple row formula |
| Parameter: Formula | ROUND(ROLLINGVAR(split_time_sc, 3), 3) |
| Parameter: Order rows by | splitId |
| Parameter: New column name | 'rvar' |

**Compute rolling computations using sample method:** These metrics compute the rolling STDEV and VAR on the current and three previous splits using the sample method:

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Multiple row formula |
| Parameter: Formula | ROUND(ROLLINGSTDEVSAMP(split_time_sc, 3), 3) |
| Parameter: Order rows by | splitId |
| Parameter: New column name | 'rstdev_samp' |

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Multiple row formula |
| Parameter: Formula | ROUND(ROLLINGVARSAMP(split_time_sc, 3), 3) |

| Parameter: Order rows by | splitId |
|---|---|
| Parameter: New column name | 'rvar_samp' |

**Results:**

When the above transforms have been completed, the results look like the following:

| lap | quarter | splitId | time_sc | split_time_sc | rvar_samp | rstdev_samp | rvar | rstdev | rmin | rmax | ravg |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 0 | l1q0 | 0 | | | | | | | | |
| 1 | 1 | l1q1 | 20.096 | 20.096 | | | 0 | 0 | 20.096 | 20.096 | 20.096 |
| 1 | 2 | l1q2 | 40.53 | 20.434 | 0.229 | 0.479 | 0.029 | 0.169 | 20.096 | 20.434 | 20.265 |
| 1 | 3 | l1q3 | 61.031 | 20.501 | 0.154 | 0.392 | 0.031 | 0.177 | 20.096 | 20.501 | 20.344 |
| 2 | 0 | l2q0 | 81.087 | 20.056 | 0.315 | 0.561 | 0.039 | 0.198 | 20.056 | 20.501 | 20.272 |
| 2 | 1 | l2q1 | 101.383 | 20.296 | 0.142 | 0.376 | 0.029 | 0.17 | 20.056 | 20.501 | 20.322 |
| 2 | 2 | l2q2 | 122.092 | 20.709 | 0.617 | 0.786 | 0.059 | 0.242 | 20.056 | 20.709 | 20.39 |
| 2 | 3 | l2q3 | 141.886 | 19.794 | 0.621 | 0.788 | 0.113 | 0.337 | 19.794 | 20.709 | 20.214 |
| 3 | 0 | l3q0 | 162.581 | 20.695 | 0.579 | 0.761 | 0.139 | 0.373 | 19.794 | 20.709 | 20.373 |
| 3 | 1 | l3q1 | 183.018 | 20.437 | 0.443 | 0.666 | 0.138 | 0.371 | 19.794 | 20.709 | 20.409 |
| 3 | 2 | l3q2 | 203.493 | 20.475 | 0.537 | 0.733 | 0.113 | 0.336 | 19.794 | 20.695 | 20.35 |
| 3 | 3 | l3q3 | 222.893 | 19.4 | 0.520 | 0.721 | 0.252 | 0.502 | 19.4 | 20.695 | 20.252 |

You can reduce the number of steps by applying a `window` transform such as the following:

| Transformation Name | Window |
|---|---|
| Parameter: Formula1 | lap |
| Parameter: Formula2 | rollingaverage(split_time_sc, 0, 3) |
| Parameter: Formula3 | rollingmax(split_time_sc, 0, 3) |
| Parameter: Formula4 | rollingmin(split_time_sc, 0, 3) |
| Parameter: Formula5 | round(rollingstdev(split_time_sc, 0, 3), 3) |
| Parameter: Formula6 | round(rollingvar(split_time_sc, 0, 3), 3) |
| Parameter: Formula7 | round(rollingstdevsamp(split_time_sc, 0, 3), 3) |
| Parameter: Formula8 | round(rollingvarsamp(split_time_sc, 0, 3), 3) |
| Parameter: Group by | lap |
| Parameter: Order by | lap |

However, you must rename all of the generated `windowX` columns.

# ROLLINGVARSAMP Function

**Contents:**

Computes the rolling variance of values forward or backward of the current row within the specified column using the sample statistical method.

- If an input value is missing or null, it is not factored in the computation. For example, for the first row in the dataset, the rolling variance of previous values is undefined.
- The row from which to extract a value is determined by the order in which the rows are organized based on the `order` parameter.
- If you are working on a randomly generated sample of your dataset, the values that you see for this function might not correspond to the values that are generated on the full dataset during job execution.
- The function takes a column name and two optional integer parameters that determine the window backward and forward of the current row.
    - The default integer parameter values are `-1` and `0`, which computes the rolling function from the current row back to the first row of the dataset.
- This function works with the Window transform. See *Window Transform*.

> **NOTE:** This function applies to a sample of the entire population. More information is below.

**Relevant terms:**

| Term | Description |
|---|---|
| Population | Population statistical functions are computed from all possible values. See *https://en.wikipedia.org/wiki/Statistical_population*. |
| Sample | Sample-based statistical functions are computed from a subset or sample of all values. See *https://en.wikipedia.org/wiki/Sampling_(statistics)*. <br><br> These function names include `SAMP` in their name. <br><br> > **NOTE:** Statistical sampling has no relationship to the samples taken within the product. When statistical functions are computed during job execution, they are applied across the entire dataset. Sample method calculations are computed at that time. |

For more information on a non-rolling version of this function, see *VAR Function*.

**Wrangle vs. SQL:** This function is part of Wrangle , a proprietary data transformation language. Wrangle is not SQL. For more information, see *Wrangle Language*.

## Basic Usage

**Column example:**

```
rollingvarsamp(myCol)
```

**Output:** Returns the rolling variance of all values in the `myCol` column from the first row of the dataset to the current one using the sample method of calculation.

**Rows before example:**

```
<span>rollingvarsamp</span>(myNumber, 100)
```

**Output:** Returns the rolling variance of the current row and the 100 previous row values in the `myNumber` column using the sample method of calculation.

**Rows before and after example:**

```
<span>rollingvarsamp</span>(myNumber, 3, 2)
```

**Output:** Returns the rolling variance of the three previous row values, the current row value, and the two rows after the current one in the `myNumber` column using the sample method of calculation.

## Syntax and Arguments

```
<span>rollingvarsamp</span>(col_ref, rowsBefore_integer, rowsAfter_integer) order:
order_col [group: group_col]
```

| Argument | Required? | Data Type | Description |
|---|---|---|---|
| col_ref | Y | string | Name of column whose values are applied to the function |
| rowsBefore_integer | N | integer | Number of rows before the current one to include in the computation |
| rowsAfter_integer | N | integer | Number of rows after the current one to include in the computation |

For more information on the `order` and `group` parameters, see *Window Transform*.

For more information on syntax standards, see *Language Documentation Syntax Notes*.

**col_ref**

Name of the column whose values you wish to use in the calculation. Column must be a numeric (Integer or Decimal) type.

- Multiple columns and wildcards are not supported.

**Usage Notes:**

| Required? | Data Type | Example Value |
|---|---|---|
| Yes | String (column reference to Integer or Decimal values) | `myColumn` |

**rowsBefore_integer, rowsAfter_integer**

Integers representing the number of rows before or after the current one from which to compute the rolling function, including the current row. For example, if the first value is `5`, the current row and the five rows before it are used in the computation. Negative values for `rowsAfter_integer` compute the rolling function from rows preceding the current one.

- `rowBefore=0` generates the current row value only.
- `rowBefore=-1` uses all rows preceding the current one.
- If `rowsAfter` is not specified, then the value `0` is applied.
- If a `group` parameter is applied, then these parameter values should be no more than the maximum number of rows in the groups.

**Usage Notes:**

| Required? | Data Type | Example Value |
|-----------|-----------|---------------|
| No | Integer | 4 |

## Examples

> **Tip:** For additional examples, see *Common Tasks*.

### Example - Rolling computations for racing splits

This example describes how to use rolling statistical functions.

**Functions**:

| Item | Description |
|------|-------------|
| ROLLINGAVERAGE Function | Computes the rolling average of values forward or backward of the current row within the specified column. |
| ROLLINGMAX Function | Computes the rolling maximum of values forward or backward of the current row within the specified column. Inputs can be Integer, Decimal, or Datetime. |
| ROLLINGSTDEV Function | Computes the rolling standard deviation of values forward or backward of the current row within the specified column. |
| ROLLINGVAR Function | Computes the rolling variance of values forward or backward of the current row within the specified column. |
| ROLLINGSTDEVSAMP Function | Computes the rolling standard deviation of values forward or backward of the current row within the specified column using the sample statistical method. |
| ROLLINGVARSAMP Function | Computes the rolling variance of values forward or backward of the current row within the specified column using the sample statistical method. |

Also:

| Item | Description |
|------|-------------|
| MERGE Function | Merges two or more columns of String type to generate output of String type. Optionally, you can insert a delimiter between the merged values. |
| ROUND Function | Rounds input value to the nearest integer. Input can be an Integer, a Decimal, a column reference, or an expression. Optional second argument can be used to specify the number of digits to which to round. |

**Source:**

In this example, the following data comes from times recorded at regular intervals during a three-lap race around a track. The source data is in cumulative time in seconds (`time_sc`). You can use ROLLING and other windowing functions to break down the data into more meaningful metrics.

| lap | quarter | time_sc |
|-----|---------|---------|
| 1 | 0 | 0.000 |

| | | |
|---|---|---|
| 1 | 1 | 19.554 |
| 1 | 2 | 39.785 |
| 1 | 3 | 60.021 |
| 2 | 0 | 80.950 |
| 2 | 1 | 101.785 |
| 2 | 2 | 121.005 |
| 2 | 3 | 141.185 |
| 3 | 0 | 162.008 |
| 3 | 1 | 181.887 |
| 3 | 2 | 200.945 |
| 3 | 3 | 220.856 |

**Transformation:**

**Primary key:** Since the quarter information repeats every lap, there is no unique identifier for each row. The following steps create this identifier:

| | |
|---|---|
| **Transformation Name** | Change column data type |
| **Parameter: Columns** | lap,quarter |
| **Parameter: New type** | String |

| | |
|---|---|
| **Transformation Name** | New formula |
| **Parameter: Formula type** | Single row formula |
| **Parameter: Formula** | MERGE(['l',lap,'q',quarter]) |
| **Parameter: New column name** | 'splitId' |

**Get split times:** Use the following transform to break down the splits for each quarter of the race:

| | |
|---|---|
| **Transformation Name** | New formula |
| **Parameter: Formula type** | Multiple row formula |
| **Parameter: Formula** | ROUND(time_sc - PREV(time_sc, 1), 3) |
| **Parameter: Order rows by** | splitId |
| **Parameter: New column name** | 'split_time_sc' |

**Compute rolling computations:** You can use the following types of computations to provide rolling metrics on the current and three previous splits:

| | |
|---|---|
| **Transformation Name** | New formula |
| **Parameter: Formula type** | Multiple row formula |
| **Parameter: Formula** | ROLLINGAVERAGE(split_time_sc, 3) |

| Parameter: Order rows by | splitId |
|---|---|
| Parameter: New column name | 'ravg' |

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Multiple row formula |
| Parameter: Formula | ROLLINGMAX(split_time_sc, 3) |
| Parameter: Order rows by | splitId |
| Parameter: New column name | 'rmax' |

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Multiple row formula |
| Parameter: Formula | ROLLINGMIN(split_time_sc, 3) |
| Parameter: Order rows by | splitId |
| Parameter: New column name | 'rmin' |

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Multiple row formula |
| Parameter: Formula | ROUND(ROLLINGSTDEV(split_time_sc, 3), 3) |
| Parameter: Order rows by | splitId |
| Parameter: New column name | 'rstdev' |

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Multiple row formula |
| Parameter: Formula | ROUND(ROLLINGVAR(split_time_sc, 3), 3) |
| Parameter: Order rows by | splitId |
| Parameter: New column name | 'rvar' |

**Compute rolling computations using sample method:** These metrics compute the rolling STDEV and VAR on the current and three previous splits using the sample method:

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Multiple row formula |
| Parameter: Formula | ROUND(ROLLINGSTDEVSAMP(split_time_sc, 3), 3) |
| Parameter: Order rows by | splitId |
| Parameter: New column | 'rstdev_samp' |

| | name | |
|---|---|---|

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Multiple row formula |
| Parameter: Formula | ROUND(ROLLINGVARSAMP(split_time_sc, 3), 3) |
| Parameter: Order rows by | splitId |
| Parameter: New column name | 'rvar_samp' |

### Results:

When the above transforms have been completed, the results look like the following:

| lap | quarter | splitId | time_sc | split_time_sc | rvar_samp | rstdev_samp | rvar | rstdev | rmin | rmax | ravg |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 0 | l1q0 | 0 | | | | | | | | |
| 1 | 1 | l1q1 | 20.096 | 20.096 | | | 0 | 0 | 20.096 | 20.096 | 20.096 |
| 1 | 2 | l1q2 | 40.53 | 20.434 | 0.229 | 0.479 | 0.029 | 0.169 | 20.096 | 20.434 | 20.265 |
| 1 | 3 | l1q3 | 61.031 | 20.501 | 0.154 | 0.392 | 0.031 | 0.177 | 20.096 | 20.501 | 20.344 |
| 2 | 0 | l2q0 | 81.087 | 20.056 | 0.315 | 0.561 | 0.039 | 0.198 | 20.056 | 20.501 | 20.272 |
| 2 | 1 | l2q1 | 101.383 | 20.296 | 0.142 | 0.376 | 0.029 | 0.17 | 20.056 | 20.501 | 20.322 |
| 2 | 2 | l2q2 | 122.092 | 20.709 | 0.617 | 0.786 | 0.059 | 0.242 | 20.056 | 20.709 | 20.39 |
| 2 | 3 | l2q3 | 141.886 | 19.794 | 0.621 | 0.788 | 0.113 | 0.337 | 19.794 | 20.709 | 20.214 |
| 3 | 0 | l3q0 | 162.581 | 20.695 | 0.579 | 0.761 | 0.139 | 0.373 | 19.794 | 20.709 | 20.373 |
| 3 | 1 | l3q1 | 183.018 | 20.437 | 0.443 | 0.666 | 0.138 | 0.371 | 19.794 | 20.709 | 20.409 |
| 3 | 2 | l3q2 | 203.493 | 20.475 | 0.537 | 0.733 | 0.113 | 0.336 | 19.794 | 20.695 | 20.35 |
| 3 | 3 | l3q3 | 222.893 | 19.4 | 0.520 | 0.721 | 0.252 | 0.502 | 19.4 | 20.695 | 20.252 |

You can reduce the number of steps by applying a `window` transform such as the following:

| Transformation Name | Window |
|---|---|
| Parameter: Formula1 | lap |
| Parameter: Formula2 | rollingaverage(split_time_sc, 0, 3) |
| Parameter: Formula3 | rollingmax(split_time_sc, 0, 3) |
| Parameter: Formula4 | rollingmin(split_time_sc, 0, 3) |
| Parameter: Formula5 | round(rollingstdev(split_time_sc, 0, 3), 3) |
| Parameter: Formula6 | round(rollingvar(split_time_sc, 0, 3), 3) |
| Parameter: Formula7 | round(rollingstdevsamp(split_time_sc, 0, 3), 3) |
| Parameter: Formula8 | round(rollingvarsamp(split_time_sc, 0, 3), 3) |
| Parameter: Group by | lap |
| Parameter: Order by | lap |

However, you must rename all of the generated `windowX` columns.

# ROLLINGCOUNTA Function

**Contents:**

Computes the rolling count of non-null values forward or backward of the current row within the specified column.

- If an input value is missing or null, it is not factored in the computation. For example, for the first row in the dataset, the rolling count of non-null values of previous values is undefined.
- The row from which to extract a value is determined by the order in which the rows are organized based on the `order` parameter.
- If you are working on a randomly generated sample of your dataset, the values that you see for this function might not correspond to the values that are generated on the full dataset during job execution.
- The function takes a column name and two optional integer parameters that determine the window backward and forward of the current row.
  - The default integer parameter values are `-1` and `0`, which computes the rolling function from the current row back to the first row of the dataset.
- This function works with the Window transform. See *Window Transform*.

For more information on a non-rolling version of this function, see *COUNTA Function.*

**Wrangle vs. SQL:** This function is part of Wrangle , a proprietary data transformation language. Wrangle is not SQL. For more information, see *Wrangle Language.*

## Basic Usage

**Column example:**

```
rollingcounta(myCol)
```

**Output:** Returns rolling count of non-null values in the `myCol` column.

**Rows before example:**

```
rollingocounta(myNumber, 3)
```

**Output:** Returns the rolling count of non-null values of the current row and the three previous row values in the `my Number` column.

**Rows before and after example:**

```
rollingcounta(myNumber, 3, 2)
```

**Output:** Returns the rolling count of non-nulls from the three previous row values, the current row value, and the two rows after the current one in the `myNumber` column.

## Syntax and Arguments

```
rollingcounta(col_ref, rowsBefore_integer, rowsAfter_integer) order: order_col [group:
group_col]
```

| Argument | Required? | Data Type | Description |
|---|---|---|---|
| col_ref | Y | string | Name of column whose values are applied to the function |
| rowsBefore_integer | N | integer | Number of rows before the current one to include in the computation |
| rowsAfter_integer | N | integer | Number of rows after the current one to include in the computation |

For more information on the `order` and `group` parameters, see *Window Transform*.

For more information on syntax standards, see *Language Documentation Syntax Notes*.

**col_ref**

Name of the column whose values are used to compute the function.

- Multiple columns and wildcards are not supported.

**Usage Notes:**

| Required? | Data Type | Example Value |
|---|---|---|
| Yes | String (column reference to Integer or Decimal values) | `myColumn` |

**rowsBefore_integer, rowsAfter_integer**

Integers representing the number of rows before or after the current one from which to compute the rolling function, including the current row. For example, if the first value is 5, the current row and the five rows before it are used in the computation. Negative values for `rowsAfter_integer` compute the rolling function from rows preceding the current one.

- `rowBefore=0` generates the current row value only.
- `rowBefore=-1` uses all rows preceding the current one.
- If `rowsAfter` is not specified, then the value `0` is applied.
- If a `group` parameter is applied, then these parameter values should be no more than the maximum number of rows in the groups.

**Usage Notes:**

| Required? | Data Type | Example Value |
|---|---|---|
| No | Integer | 4 |

Examples

> **Tip:** For additional examples, see *Common Tasks*.

**Example - Counting messages**

In the following example, messages are tabulated every 10 seconds from a system. If no message is generated, null values are returned.

**Source:**

| Timestamp | msgType | msgDescription |
|---|---|---|
| 15:10:00 PM | warning | Server restarted. |
| 15:10:10 PM | warning | Unable to locate patterns file. |
| 15:10:20 PM | | |
| 15:10:30 PM | | |
| 15:10:40 PM | error | Cannot connect to data source. |
| 15:10:50 PM | error | Cannot open dataset. |
| 15:11:00 PM | | |
| 15:11:10 PM | | |
| 15:11:20 PM | error | Insufficient permissions to write to target location. |
| 15:11:30 PM | | |
| 15:11:40 PM | warning | Server restarted. |
| 15:11:50 PM | warning | Unable to locate patterns file. |
| 15:12:00 PM | error | Data node offline. |
| 15:12:10 PM | | |
| 15:12:20 PM | | |
| 15:12:30 PM | warning | Invalid statement in recipe. |
| 15:12:40 PM | | |
| 15:12:50 PM | | |

**Transformation:**

You are interested in counting the number of entries for the preceding minute for each row. You add the following:

| Transformation Name | `New formula` |
|---|---|
| Parameter: Formula type | `Multiple row formula` |
| Parameter: Sort rows by | `Timestamp` |
| Parameter: Formula | `rollingcounta(msgType, 5, 0)` |
| Parameter: New column name | `'rollingcounta_msgType'` |

**Results:**

| Timestamp | msgType | msgDescription | rollingcounta_msgType |
|---|---|---|---|
| 15:10:00 PM | warning | Server restarted. | 1 |
| 15:10:10 PM | warning | Unable to locate patterns file. | 2 |
| 15:10:20 PM | | | 2 |
| 15:10:30 PM | | | 2 |
| | | | |

| | | | |
|---|---|---|---|
| 15:10:40 PM | error | Cannot connect to data source. | 3 |
| 15:10:50 PM | error | Cannot open dataset. | 4 |
| 15:11:00 PM | | | 3 |
| 15:11:10 PM | | | 2 |
| 15:11:20 PM | error | Insufficient permissions to write to target location. | 3 |
| 15:11:30 PM | | | 3 |
| 15:11:40 PM | warning | Server restarted. | 3 |
| 15:11:50 PM | warning | Unable to locate patterns file. | 3 |
| 15:12:00 PM | error | Data node offline. | 4 |
| 15:12:10 PM | | | 4 |
| 15:12:20 PM | | | 3 |
| 15:12:30 PM | warning | Invalid statement in recipe. | 4 |
| 15:12:40 PM | | | 3 |
| 15:12:50 PM | | | 2 |

# ROLLINGKTHLARGEST Function

**Contents:**

Computes the rolling *kth* largest value forward or backward of the current row. Inputs can be Integer, Decimal, or Datetime.

`KTHLARGEST` extracts the ranked value from the values in a column, where `k=1` returns the maximum value. The value for `k` must be between 1 and 1000, inclusive. For purposes of this calculation, two instances of the same value are treated as separate values. So, if your dataset contains three rows with column values `10`, `9`, and `9`, then `KTHLARGEST` returns `9` for `k=2` and `k=3`.

`ROLLINGKTHLARGEST` computes the `KTHLARGEST` value across a defined window of values within a column.

- If an input value is missing or null, it is not factored in the computation. For example, for the first row in the dataset, the rolling calculation of previous values is undefined.
- The row from which to extract a value is determined by the order in which the rows are organized based on the `order` parameter.
- If you are working on a randomly generated sample of your dataset, the values that you see for this function might not correspond to the values that are generated on the full dataset during job execution.
- Inputs:
    - Required column name
    - Required *kth* value, which is a positive integer
    - Two optional integer parameters that determine the window backward and forward of the current row. The default integer parameter values are `-1` and `0`, which computes the rolling function from the current row back to the first row of the dataset.
- This function works with the Window transform. See *Window Transform*.

For more information on a non-rolling version of this function, see *KTHLARGEST Function*.

**Wrangle vs. SQL:** This function is part of Wrangle , a proprietary data transformation language. Wrangle is not SQL. For more information, see *Wrangle Language*.

## Basic Usage

**Column example:**

```
rollingkthlargest(myCol, 2)
```

**Output:** Returns the rolling second largest of all values in the `myCol` column.

**Rows before example:**

```
rollingkthlargest(myNumber, 2, 3)
```

**Output:** Returns the rolling second largest value of the current row and the two previous row values in the `myNumber` column.

**Rows before and after example:**

```
rollingkthlargest(myNumber, 4, 3, 2)
```

**Output:** Returns the rolling fourth largest value of the two previous row values, the current row value, and the two rows after the current one in the `myNumber` column.

## Syntax and Arguments

```
rollingkthlargest(col_ref, rowsBefore_integer, rowsAfter_integer) order: order_col [group:
group_col]
```

| Argument | Required? | Data Type | Description |
|---|---|---|---|
| col_ref | Y | string | Name of column whose values are applied to the function |
| k_integer | Y | integer (positive) | The ranking of the value to extract from the source column |
| rowsBefore_integer | N | integer | Number of rows before the current one to include in the computation |
| rowsAfter_integer | N | integer | Number of rows after the current one to include in the computation |

For more information on the `order` and `group` parameters, see *Window Transform*.

For more information on syntax standards, see *Language Documentation Syntax Notes*.

**col_ref**

Name of the column whose values are used to compute the function. Inputs must be Integer, Decimal, or Datetime values.

> **NOTE:** If the input is in Datetime type, the output is in unixtime format. You can wrap these outputs in the DATEFORMAT function to output the results in the appropriate Datetime format. See *DATEFORMAT Function*.

- Multiple columns and wildcards are not supported.

**Usage Notes:**

| Required? | Data Type | Example Value |
|---|---|---|
| Yes | String (column reference to Integer or Decimal values) | `myColumn` |

**k_integer**

Integer representing the ranking of the unique value to extract from the source column. Duplicate values are treated as separate values for purposes of this function's calculation.

> **NOTE:** The value for `k` must be an integer between 1 and 1,000 inclusive.

- `k=1` represents the maximum value in the column.

- If k is greater than or equal to the number of values in the column, the minimum value is returned.
- Missing and null values are not factored into the ranking of `k`.

**Usage Notes:**

| Required? | Data Type | Example Value |
|-----------|-----------|---------------|
| Yes | Integer (positive) | 4 |

### rowsBefore_integer, rowsAfter_integer

Integers representing the number of rows before or after the current one from which to compute the rolling function, including the current row. For example, if the first value is 5, the current row and the four rows after it are used in the computation. Negative values for `rowsAfter_integer` compute the rolling function from rows preceding the current one.

- `rowBefore=1` generates the current row value only.
- `rowBefore=-1` uses all rows preceding the current one.
- If `rowsAfter` is not specified, then the value `0` is applied.
- If a `group` parameter is applied, then these parameter values should be no more than the maximum number of rows in the groups.

**Usage Notes:**

| Required? | Data Type | Example Value |
|-----------|-----------|---------------|
| No | Integer | 4 |

## Examples

> **Tip:** For additional examples, see *Common Tasks*.

### Example - ROLLINGKTHLARGEST functions

This example describes how to use rolling kthlargest functions for calculating ranking of values within a defined window of rows.

**Functions**:

| Item | Description |
|------|-------------|
| ROLLINGKTHLARGEST Function | Computes the rolling *kth* largest value forward or backward of the current row. Inputs can be Integer, Decimal, or Datetime. |
| ROLLINGKTHLARGESTUNIQUE Function | Computes the rolling unique *kth* largest value forward or backward of the current row. Inputs can be Integer, Decimal, or Datetime. |
| ROWNUMBER Function | Generates a new column containing the row number as sorted by the `order` parameter and optionally grouped by the `group` parameter. |

The following dataset contains daily counts of server restarts across three servers over the preceding week. High server restart counts can indicate poor server health. In this example, you are interested in knowing for each server the rolling highest and second highest count of restarts per server over the previous week.

**Source:**

| Date | Server | Restarts |
|---|---|---|
| 2/21/18 | s01 | 4 |
| 2/21/18 | s02 | 0 |
| 2/21/18 | s03 | 0 |
| 2/22/18 | s01 | 4 |
| 2/22/18 | s02 | 1 |
| 2/22/18 | s03 | 2 |
| 2/23/18 | s01 | 2 |
| 2/23/18 | s02 | 3 |
| 2/23/18 | s03 | 4 |
| 2/24/18 | s01 | 1 |
| 2/24/18 | s02 | 0 |
| 2/24/18 | s03 | 2 |
| 2/25/18 | s01 | 5 |
| 2/25/18 | s02 | 0 |
| 2/25/18 | s03 | 4 |
| 2/26/18 | s01 | 1 |
| 2/26/18 | s02 | 2 |
| 2/26/18 | s03 | 1 |
| 2/27/18 | s01 | 1 |
| 2/27/18 | s02 | 2 |
| 2/27/18 | s03 | 2 |

**Transformation:**

First, you want to maintain the row information as a separate column. Since data is ordered already by the `Date` column, you can use the following:

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | ROWNUMBER() |
| Parameter: New column name | 'entryId' |

Use the following function to compute the rolling *kth* largest value of server restarts per server over the previous week. In this case, you can use the `ROLLINGKTHLARGEST` function, setting *k*=1. Uniqueness doesn't matter for calculating the highest value:

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Multiple row formula |
| Parameter: Formula | rollingkthlargest(Restarts, 1, 6, 0) |

| | |
|---|---|
| **Parameter: Sort Rows by** | Server |
| **Parameter: Group Rows by** | Server |
| **Parameter: New column name** | 'rollingkthlargest_1' |

Use the following function to compute the rolling second highest value. In this case, you can use ROLLINGKTHLA RGESTUNIQUE:

| | |
|---|---|
| **Transformation Name** | New formula |
| **Parameter: Formula type** | Multiple row formula |
| **Parameter: Formula** | rollingkthlargestunique(Restarts, 2, 6, 0) |
| **Parameter: Sort Rows by** | Server |
| **Parameter: Group Rows by** | Server |
| **Parameter: New column name** | 'rollingKthLargestUnique_2' |

**Results:**

| entryId | Date | Server | Restarts | rollingKthLargestUnique_2 | rollingkthlargest_Restarts |
|---|---|---|---|---|---|
| 3 | 2/21/18 | s02 | 0 | 0 | 0 |
| 6 | 2/22/18 | s02 | 1 | 0 | 1 |
| 9 | 2/23/18 | s02 | 3 | 1 | 3 |
| 12 | 2/24/18 | s02 | 0 | 1 | 3 |
| 15 | 2/25/18 | s02 | 0 | 1 | 3 |
| 18 | 2/26/18 | s02 | 2 | 2 | 3 |
| 21 | 2/27/18 | s02 | 2 | 2 | 3 |
| 4 | 2/21/18 | s03 | 0 | 0 | 0 |
| 7 | 2/22/18 | s03 | 2 | 0 | 2 |
| 10 | 2/23/18 | s03 | 4 | 2 | 4 |
| 13 | 2/24/18 | s03 | 2 | 2 | 4 |
| 16 | 2/25/18 | s03 | 4 | 2 | 4 |
| 19 | 2/26/18 | s03 | 1 | 2 | 4 |
| 22 | 2/27/18 | s03 | 2 | 2 | 4 |
| 2 | 2/21/18 | s01 | 4 | 4 | 4 |
| 5 | 2/22/18 | s01 | 4 | 4 | 4 |
| 8 | 2/23/18 | s01 | 2 | 2 | 4 |
| 11 | 2/24/18 | s01 | 1 | 2 | 4 |
| 14 | 2/25/18 | s01 | 5 | 4 | 5 |
| 17 | 2/26/18 | s01 | 1 | 4 | 5 |
| 20 | 2/27/18 | s01 | 1 | 4 | 5 |

# ROLLINGKTHLARGESTUNIQUE Function

**Contents:**

---

Computes the rolling unique *kth* largest value forward or backward of the current row. Inputs can be Integer, Decimal, or Datetime.

For purposes of this calculation, two instances of the same value are treated at one value for k. So, if your dataset contains four rows with column values `10`, `9`, `9`, and `8`, then `KTHLARGESTUNIQUE` returns `9` for `k=2` and `8` for `k=3`.

`ROLLINGKTHLARGESTUNIQUE` computes the `KTHLARGESTUNIQUE` value across a defined window of values within a column.

- If an input value is missing or null, it is not factored in the computation. For example, for the first row in the dataset, the rolling calculation of previous values is undefined.
- The row from which to extract a value is determined by the order in which the rows are organized based on the `order` parameter.
- If you are working on a randomly generated sample of your dataset, the values that you see for this function might not correspond to the values that are generated on the full dataset during job execution.
- Inputs:
    - Required column name
    - Required *kth* value, which is a positive integer
    - Two optional integer parameters that determine the window backward and forward of the current row. The default integer parameter values are `-1` and `0`, which computes the rolling function from the current row back to the first row of the dataset.
- This function works with the Window transform. See *Window Transform*.

For more information on a non-rolling version of this function, see *KTHLARGESTUNIQUE Function*.

**Wrangle vs. SQL:** This function is part of Wrangle , a proprietary data transformation language. Wrangle is not SQL. For more information, see *Wrangle Language*.

## Basic Usage

**Column example:**

```
rollingkthlargestunique(myCol, 2)
```

**Output:** Returns the rolling second largest unique value in the `myCol` column from the first row of the dataset to the current one.

**Rows before example:**

```
rollingkthlargestunique(myNumber, 2, 3)
```

**Output:** Returns the rolling second largest unique value of the current row and the two previous row values in the `myNumber` column.

**Rows before and after example:**

```
rollingkthlargestunique(myNumber, 4, 3, 2)
```

**Output:** Returns the rolling fourth largest unique value of the two previous row values, the current row value, and the two rows after the current one in the `myNumber` column.

## Syntax and Arguments

```
rollingkthlargestunique(col_ref, rowsBefore_integer, rowsAfter_integer) order: order_col
[group: group_col]
```

| Argument | Required? | Data Type | Description |
|---|---|---|---|
| col_ref | Y | string | Name of column whose values are applied to the function |
| k_integer | Y | integer (positive) | The ranking of the unique value to extract from the source column |
| rowsBefore_integer | N | integer | Number of rows before the current one to include in the computation |
| rowsAfter_integer | N | integer | Number of rows after the current one to include in the computation |

For more information on the `order` and `group` parameters, see *Window Transform*.

For more information on syntax standards, see *Language Documentation Syntax Notes*.

**col_ref**

Name of the column whose values are used to compute the function. Inputs must be Integer, Decimal, or Datetime values.

> **NOTE:** If the input is in Datetime type, the output is in unixtime format. You can wrap these outputs in the DATEFORMAT function to output the results in the appropriate Datetime format. See *DATEFORMAT Function*.

- Multiple columns and wildcards are not supported.

**Usage Notes:**

| Required? | Data Type | Example Value |
|---|---|---|
| Yes | String (column reference) | `myColumn` |

**k_integer**

Integer representing the ranking of the unique value to extract from the source column. Duplicate values are treated as a single value for purposes of this function's calculation.

> **NOTE:** The value for `k` must be an integer between 1 and 1,000 inclusive.

- `k=1` represents the maximum value in the column.
- If k is greater than or equal to the number of values in the column, the minimum value is returned.

- Missing and null values are not factored into the ranking of `k`.

**Usage Notes:**

| Required? | Data Type | Example Value |
|---|---|---|
| Yes | Integer (positive) | 4 |

### rowsBefore_integer, rowsAfter_integer

Integers representing the number of rows before or after the current one from which to compute the rolling function, including the current row. For example, if the first value is `5`, the current row and the four rows after it are used in the computation. Negative values for `rowsAfter_integer` compute the rolling function from rows preceding the current one.

- `rowBefore=1` generates the current row value only.
- `rowBefore=-1` uses all rows preceding the current one.
- If `rowsAfter` is not specified, then the value `0` is applied.
- If a `group` parameter is applied, then these parameter values should be no more than the maximum number of rows in the groups.

**Usage Notes:**

| Required? | Data Type | Example Value |
|---|---|---|
| No | Integer | 4 |

## Examples

> **Tip:** For additional examples, see *Common Tasks*.

### Example - ROLLINGKTHLARGEST functions

This example describes how to use rolling kthlargest functions for calculating ranking of values within a defined window of rows.

**Functions**:

| Item | Description |
|---|---|
| ROLLINGKTHLARGEST Function | Computes the rolling *kth* largest value forward or backward of the current row. Inputs can be Integer, Decimal, or Datetime. |
| ROLLINGKTHLARGESTUNIQUE Function | Computes the rolling unique *kth* largest value forward or backward of the current row. Inputs can be Integer, Decimal, or Datetime. |
| ROWNUMBER Function | Generates a new column containing the row number as sorted by the `order` parameter and optionally grouped by the `group` parameter. |

The following dataset contains daily counts of server restarts across three servers over the preceding week. High server restart counts can indicate poor server health. In this example, you are interested in knowing for each server the rolling highest and second highest count of restarts per server over the previous week.

**Source:**

| Date | Server | Restarts |
|------|--------|----------|
| 2/21/18 | s01 | 4 |
| 2/21/18 | s02 | 0 |
| 2/21/18 | s03 | 0 |
| 2/22/18 | s01 | 4 |
| 2/22/18 | s02 | 1 |
| 2/22/18 | s03 | 2 |
| 2/23/18 | s01 | 2 |
| 2/23/18 | s02 | 3 |
| 2/23/18 | s03 | 4 |
| 2/24/18 | s01 | 1 |
| 2/24/18 | s02 | 0 |
| 2/24/18 | s03 | 2 |
| 2/25/18 | s01 | 5 |
| 2/25/18 | s02 | 0 |
| 2/25/18 | s03 | 4 |
| 2/26/18 | s01 | 1 |
| 2/26/18 | s02 | 2 |
| 2/26/18 | s03 | 1 |
| 2/27/18 | s01 | 1 |
| 2/27/18 | s02 | 2 |
| 2/27/18 | s03 | 2 |

**Transformation:**

First, you want to maintain the row information as a separate column. Since data is ordered already by the `Date` column, you can use the following:

| Transformation Name | New formula |
|---------------------|-------------|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | ROWNUMBER() |
| Parameter: New column name | 'entryId' |

Use the following function to compute the rolling *kth* largest value of server restarts per server over the previous week. In this case, you can use the `ROLLINGKTHLARGEST` function, setting *k*=1. Uniqueness doesn't matter for calculating the highest value:

| Transformation Name | New formula |
|---------------------|-------------|
| Parameter: Formula type | Multiple row formula |
| Parameter: Formula | rollingkthlargest(Restarts, 1, 6, 0) |
| | |

| | |
|---|---|
| **Parameter: Sort Rows by** | `Server` |
| **Parameter: Group Rows by** | `Server` |
| **Parameter: New column name** | `'rollingkthlargest_1'` |

Use the following function to compute the rolling second highest value. In this case, you can use `ROLLINGKTHLA RGESTUNIQUE`:

| | |
|---|---|
| **Transformation Name** | `New formula` |
| **Parameter: Formula type** | `Multiple row formula` |
| **Parameter: Formula** | `rollingkthlargestunique(Restarts, 2, 6, 0)` |
| **Parameter: Sort Rows by** | `Server` |
| **Parameter: Group Rows by** | `Server` |
| **Parameter: New column name** | `'rollingKthLargestUnique_2'` |

**Results:**

| entryId | Date | Server | Restarts | rollingKthLargestUnique_2 | rollingkthlargest_Restarts |
|---|---|---|---|---|---|
| 3 | 2/21/18 | s02 | 0 | 0 | 0 |
| 6 | 2/22/18 | s02 | 1 | 0 | 1 |
| 9 | 2/23/18 | s02 | 3 | 1 | 3 |
| 12 | 2/24/18 | s02 | 0 | 1 | 3 |
| 15 | 2/25/18 | s02 | 0 | 1 | 3 |
| 18 | 2/26/18 | s02 | 2 | 2 | 3 |
| 21 | 2/27/18 | s02 | 2 | 2 | 3 |
| 4 | 2/21/18 | s03 | 0 | 0 | 0 |
| 7 | 2/22/18 | s03 | 2 | 0 | 2 |
| 10 | 2/23/18 | s03 | 4 | 2 | 4 |
| 13 | 2/24/18 | s03 | 2 | 2 | 4 |
| 16 | 2/25/18 | s03 | 4 | 2 | 4 |
| 19 | 2/26/18 | s03 | 1 | 2 | 4 |
| 22 | 2/27/18 | s03 | 2 | 2 | 4 |
| 2 | 2/21/18 | s01 | 4 | 4 | 4 |
| 5 | 2/22/18 | s01 | 4 | 4 | 4 |
| 8 | 2/23/18 | s01 | 2 | 2 | 4 |
| 11 | 2/24/18 | s01 | 1 | 2 | 4 |
| 14 | 2/25/18 | s01 | 5 | 4 | 5 |
| 17 | 2/26/18 | s01 | 1 | 4 | 5 |
| 20 | 2/27/18 | s01 | 1 | 4 | 5 |

# ROLLINGLIST Function

**Contents:**

Computes the rolling list of values forward or backward of the current row within the specified column and returns an array of these values.

- If an input value is missing or null, it is not factored in the computation. For example, for the first row in the dataset, the rolling count of distinct values of previous values is undefined.
- The row from which to extract a value is determined by the order in which the rows are organized based on the `order` parameter.
- If you are working on a randomly generated sample of your dataset, the values that you see for this function might not correspond to the values that are generated on the full dataset during job execution.
- The function takes a column name and three optional integer parameters that determine the maximum number of values and the window backward and forward of the current row.
    - By default, the list is limited to `1000` values. To change the maximum number of values, specify a value for the `limit` parameter.
    - For the window parameters, the default values are `-1` and `0`, which computes the rolling function from the current row back to the first row of the dataset.
- This function works with the Window transform. See *Window Transform*.

For more information on a non-rolling version of this function, see *LIST Function*.

**Wrangle vs. SQL:** This function is part of Wrangle , a proprietary data transformation language. Wrangle is not SQL. For more information, see *Wrangle Language*.

## Basic Usage

**Column example:**

```
rollinglist(myCol)
```

**Output:** Returns the rolling list of values in the `myCol` column from the first row of the dataset to the current one.

**Rows before example:**

```
rollinglist(myNumber, 5, 20)
```

**Output:** Returns the rolling list of values of the current row and the twenty previous row values in the `myNumber` column, with a limit of 5 total values.

**Rows before and after example:**

```
rollinglist(myNumber, 20, 299, 200)
```

**Output:** Returns the rolling list of values from the previous 299 rows, the current row value, and the 200 rows after the current one in the `myNumber` column, with a limit of 20 total values.

## Syntax and Arguments

```
rollinglist(col_ref, rowsBefore_integer, rowsAfter_integer) order: order_col [group:
group_col]
```

| Argument | Required? | Data Type | Description |
|---|---|---|---|
| col_ref | Y | string | Name of column whose values are applied to the function |
| limit_int | N | integer | Maximum number of values to extract into the list array. From $1$ to $1000$. |
| rowsBefore_integer | N | integer | Number of rows before the current one to include in the computation |
| rowsAfter_integer | N | integer | Number of rows after the current one to include in the computation |

For more information on the `order` and `group` parameters, see *Window Transform*.

For more information on syntax standards, see *Language Documentation Syntax Notes*.

**col_ref**

Name of the column whose values are used to compute the function.

* Multiple columns and wildcards are not supported.

**Usage Notes:**

| Required? | Data Type | Example Value |
|---|---|---|
| Yes | String (column reference to Integer or Decimal values) | `myColumn` |

**limit_int**

Non-negative integer that defines the maximum number of values to extract into the list array.

> **NOTE:** If specified, this value must between 1 and 1000, inclusive.

> **NOTE:** Do not use the limiting argument in a `LIST` function call on a flat aggregate, in which all values in a column have been inserted into a single cell. In this case, you might be able to use the limit argument if you also specify a `group` parameter. Misuse of the `LIST` function can cause the application to crash.

**Usage Notes:**

| Required? | Data Type | Example Value |
|---|---|---|
| No | Integer | `50` |

**rowsBefore_integer, rowsAfter_integer**

Integers representing the number of rows before or after the current one from which to compute the rolling function, including the current row. For example, if the first value is `5`, the current row and the four rows after it are used in the computation. Negative values for `rowsAfter_integer` compute the rolling function from rows preceding the current one.

- `rowBefore=1` generates the current row value only.
- `rowBefore=-1` uses all rows preceding the current one.
- If `rowsAfter` is not specified, then the value `0` is applied.
- If a `group` parameter is applied, then these parameter values should be no more than the maximum number of rows in the groups.

**Usage Notes:**

| Required? | Data Type | Example Value |
|---|---|---|
| No | Integer | 4 |

## Examples

> **Tip:** For additional examples, see *Common Tasks*.

**Example - Recent Finishers by Boat Type**

The following dataset includes the finishing times for each boat in a race. As part of the race, each boat may be assigned one or more penalties in terms of seconds. So, the total time for the race is computed by adding two columns.

You are interested in the list of recent finishers by each finishing time.

**Source:**

| id | pilotName | boatType | raceTime | racePenalties |
|---|---|---|---|---|
| 1 | Schmidt | Sunfish | 4573.8 | 53 |
| 2 | Bolt | Laser | 4934.21 | 11 |
| 3 | Masters | Force 5 | 4446.89 | 70 |
| 4 | Jamison | Force 5 | 4355.79 | 31 |
| 5 | Williams | Sunfish | 4675.86 | 15 |
| 6 | Hobart | Laser | 5077.5 | 50 |
| 7 | Millingham | Laser | 4940.09 | 54 |
| 8 | Nelson | Force 5 | 5116.14 | 56 |
| 9 | Greene | Sunfish | 5105.94 | 5 |
| 10 | Danielson | Laser | 4964.03 | 18 |
| 11 | Cooper | Force 5 | 5281.55 | 13 |
| 12 | Stevens | Laser | 5176.35 | 0 |
| 13 | Young | Sunfish | 5038.11 | 16 |
| 14 | Thompson | Force 5 | 5252.9 | 62 |

| 15 | McDonald | Laser | 5052.24 | 20 |
|----|----------|-------|---------|----|
| 16 | O'Roarke | Sunfish | 5080.76 | 45 |
| 17 | Collins | Sunfish | 5176.09 | 10 |
| 18 | Wright | Laser | 5391.61 | 34 |
| 19 | Black | Sunfish | 5023.32 | 32 |
| 20 | Bush | Force 5 | 5200.37 | 28 |

### Transformation:

Compute the total time for each racer by summing the `raceTime` column and the `racePenalties` column:

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: raceTime + racePenalties | FCTN(input) |
| Parameter: New column name | 'totalRaceTime' |

You can then compute the list of recent finishers by boat type, automatically sorting the generated lists by the `totalRaceTime` column:

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Multiple row formula |
| Parameter: Formula | rollinglist(boatType, 10, 4, 0) |
| Parameter: Order rows by | totalRaceTime |
| Parameter: New column name | 'last5FinisherBoatTypes' |

### Results:

| id | pilotName | boatType | last5FinisherBoatTypes | raceTime | racePenalties | totalRaceTime |
|----|-----------|----------|------------------------|----------|---------------|---------------|
| 4 | Jamison | Force 5 | ["Force 5"] | 4355.79 | 31 | 4386.79 |
| 3 | Masters | Force 5 | ["Force 5","Force 5"] | 4446.89 | 70 | 4516.89 |
| 1 | Schmidt | Sunfish | ["Force 5","Force 5","Sunfish"] | 4573.8 | 53 | 4626.8 |
| 5 | Williams | Sunfish | ["Force 5","Force 5","Sunfish","Sunfish"] | 4675.86 | 15 | 4690.86 |
| 2 | Bolt | Laser | ["Force 5","Force 5","Sunfish","Sunfish","Laser"] | 4934.21 | 11 | 4945.21 |
| 10 | Danielson | Laser | ["Force 5","Sunfish","Sunfish","Laser","Laser"] | 4964.03 | 18 | 4982.03 |
| 7 | Millingham | Laser | ["Sunfish","Sunfish","Laser","Laser","Laser"] | 4940.09 | 54 | 4994.09 |
| 13 | Young | Sunfish | ["Sunfish","Laser","Laser","Laser","Sunfish"] | 5038.11 | 16 | 5054.11 |
| 19 | Black | Sunfish | ["Laser","Laser","Laser","Sunfish","Sunfish"] | 5023.32 | 32 | 5055.32 |
| 15 | McDonald | Laser | ["Laser","Laser","Sunfish","Sunfish","Laser"] | 5052.24 | 20 | 5072.24 |
| 9 | Greene | Sunfish | ["Laser","Sunfish","Sunfish","Laser","Sunfish"] | 5105.94 | 5 | 5110.94 |
| 16 | O'Roarke | Sunfish | ["Sunfish","Sunfish","Laser","Sunfish","Sunfish"] | 5080.76 | 45 | 5125.76 |
| 6 | Hobart | Laser | ["Sunfish","Laser","Sunfish","Sunfish","Laser"] | 5077.5 | 50 | 5127.5 |

| 8 | Nelson | Force 5 | ["Laser","Sunfish","Sunfish","Laser","Force 5"] | 5116.14 | 56 | 5172.14 |
|---|---|---|---|---|---|---|
| 12 | Stevens | Laser | ["Sunfish","Sunfish","Laser","Force 5","Laser"] | 5176.35 | 0 | 5176.35 |
| 17 | Collins | Sunfish | ["Sunfish","Laser","Force 5","Laser","Sunfish"] | 5176.09 | 10 | 5186.09 |
| 20 | Bush | Force 5 | ["Laser","Force 5","Laser","Sunfish","Force 5"] | 5200.37 | 28 | 5228.37 |
| 11 | Cooper | Force 5 | ["Force 5","Laser","Sunfish","Force 5","Force 5"] | 5281.55 | 13 | 5294.55 |
| 14 | Thompson | Force 5 | ["Laser","Sunfish","Force 5","Force 5","Force 5"] | 5252.9 | 62 | 5314.9 |
| 18 | Wright | Laser | ["Sunfish","Force 5","Force 5","Force 5","Laser"] | 5391.61 | 34 | 5425.61 |

# ROWNUMBER Function

Generates a new column containing the row number as sorted by the `order` parameter and optionally grouped by the `group` parameter.

> **Tip:** To generate row identifiers by the original order in the source data, use the `$sourcerownumber` reference. See *Source Metadata References*.

This function works with the following transforms:

- *Window Transform*
- *Set Transform*
- *Derive Transform*

**Wrangle vs. SQL:** This function is part of Wrangle , a proprietary data transformation language. Wrangle is not SQL. For more information, see *Wrangle Language*.


## Basic Usage

**Example:**

```
rownumber()
```

**Output:** Returns the row number of each row.

**Example with grouping:**

```
rownumber() order:Date group:QTR
```

**Output:** Returns the row number of each row as ordered by the values in the `Date` column grouped by the `QTR` v alues. For each quarter value, the row number counter resets.

## Syntax and Arguments

```
rownumber() order: order_col [group: group_col]
```

For more information on the `order` and `group` parameters, see *Window Transform*.

For more information on syntax standards, see *Language Documentation Syntax Notes*.

## Examples

> **Tip:** For additional examples, see *Common Tasks*.

**Example - Rolling window functions**

This example describes how to use rolling computational functions.

**Functions**:

| Item | Description |
|------|-------------|

| | |
|---|---|
| ROLLINGSUM Function | Computes the rolling sum of values forward or backward of the current row within the specified column. |
| ROLLINGAVERAGE Function | Computes the rolling average of values forward or backward of the current row within the specified column. |
| ROWNUMBER Function | Generates a new column containing the row number as sorted by the `order` parameter and optionally grouped by the `group` parameter. |

Also:

| Item | Description |
|---|---|
| MONTH Function | Derives the month integer value from a Datetime value. Source value can be a a reference to a column containing Datetime values or a literal. |
| FLOOR Function | Computes the largest integer that is not more than the input value. Input can be an Integer, a Decimal, a column reference, or an expression. |
| ROWNUMBER Function | Generates a new column containing the row number as sorted by the `order` parameter and optionally grouped by the `group` parameter. |

The following dataset contains sales data over the final quarter of the year.

**Source:**

| Date | Sales |
|---|---|
| 10/2/16 | 200 |
| 10/9/16 | 500 |
| 10/16/16 | 350 |
| 10/23/16 | 400 |
| 10/30/16 | 190 |
| 11/6/16 | 550 |
| 11/13/16 | 610 |
| 11/20/16 | 480 |
| 11/27/16 | 660 |
| 12/4/16 | 690 |
| 12/11/16 | 810 |
| 12/18/16 | 950 |
| 12/25/16 | 1020 |
| 1/1/17 | 680 |

**Transformation:**

First, you want to maintain the row information as a separate column. Since data is ordered already by the `Date` column, you can use the following:

| | |
|---|---|
| **Transformation Name** | `Window` |
| **Parameter: Formulas** | `ROWNUMBER()` |
| **Parameter: Order by** | `Date` |

Rename this column to `rowId` for week of quarter.

Now, you want to extract month and week information from the `Date` values. Deriving the month value:

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | MONTH(Date) |
| Parameter: New column name | 'Month' |

Deriving the quarter value:

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | (1 + FLOOR(((month-1)/3))) |
| Parameter: New column name | 'QTR' |

Deriving the week-of-quarter value:

| Transformation Name | Window |
|---|---|
| Parameter: Formulas | ROWNUMBER() |
| Parameter: Group by | QTR |
| Parameter: Order by | Date |

Rename this column `WOQ` (week of quarter).

Deriving the week-of-month value:

| Transformation Name | Window |
|---|---|
| Parameter: Formulas | ROWNUMBER() |
| Parameter: Group by | Month |
| Parameter: Order by | Date |

Rename this column `WOM` (week of month).

Now, you perform your rolling computations. Compute the running total of sales using the following:

| Transformation Name | Window |
|---|---|
| Parameter: Formulas | ROLLINGSUM(Sales, -1, 0) |
| Parameter: Group by | QTR |
| Parameter: Order by | Date |

The `-1` parameter is used in the above computation to gather the rolling sum of all rows of data from the current one to the first one. Note that the use of the `QTR` column for grouping, which moves the value for the `01/01/2017` into its own computational bucket. This may or may not be preferred.

Rename this column `QTD` (quarter to-date). Now, generate a similar column to compute the rolling average of weekly sales for the quarter:

| | |
|---|---|
| **Transformation Name** | `Window` |
| **Parameter: Formulas** | `ROUND(ROLLINGAVERAGE(Sales, -1, 0))` |
| **Parameter: Group by** | `QTR` |
| **Parameter: Order by** | `Date` |

Since the `ROLLINGAVERAGE` function can compute fractional values, it is wrapped in the `ROUND` function for neatness. Rename this column `avgWeekByQuarter`.

**Results:**

When the unnecessary columns are dropped and some reordering is applied, your dataset should look like the following:

| Date | WOQ | Sales | QTD | avgWeekByQuarter |
|---|---|---|---|---|
| 10/2/16 | 1 | 200 | 200 | 200 |
| 10/9/16 | 2 | 500 | 700 | 350 |
| 10/16/16 | 3 | 350 | 1050 | 350 |
| 10/23/16 | 4 | 400 | 1450 | 363 |
| 10/30/16 | 5 | 190 | 1640 | 328 |
| 11/6/16 | 6 | 550 | 2190 | 365 |
| 11/13/16 | 7 | 610 | 2800 | 400 |
| 11/20/16 | 8 | 480 | 3280 | 410 |
| 11/27/16 | 9 | 660 | 3940 | 438 |
| 12/4/16 | 10 | 690 | 4630 | 463 |
| 12/11/16 | 11 | 810 | 5440 | 495 |
| 12/18/16 | 12 | 950 | 6390 | 533 |
| 12/25/16 | 13 | 1020 | 7410 | 570 |
| 1/1/17 | 1 | 680 | 680 | 680 |

# SESSION Function

**Contents:**

Generates a new session identifier based on a sorted column of timestamps and a specified rolling timeframe.

The `SESSION` function takes three parameters:

- reference to column containing Datetime values used to identify sessions.
- Numeric value to identify the length of the rolling timeframe that demarcates a session.
- Unit of measure for the length of the rolling timeframe.

Like other windowing, `order` and `group` parameters can be applied. You can use the `group` and `order` paramet ers to define the groups of records and the order of those records to which this transform is applied.

> **NOTE:** While not explicitly required, you should use the `order` parameter to define a column used to sort the dataset rows. This column should match the column reference in the `SESSION` function.

In the generated column, identifiers are assigned to each row based on the computed session to which the row data belongs. Session IDs begin at `1` and increment, within each group.

- For each new group, session identifiers begin with the value `1`. As a result, session identifiers are unique only within the group, and the combination of group identifier and session identifier is a unique key within the dataset.

This function works with the following transforms:

- *Window Transform*
- *Set Transform*
- *Derive Transform*

**Wrangle vs. SQL:** This function is part of Wrangle , a proprietary data transformation language. Wrangle is not SQL. For more information, see *Wrangle Language*.

## Basic Usage

```
session(Timestamp, 1, hour) order:&apos;Timestamp&apos;
```

**Output:** Returns session identifiers for groups of rows based on 1-hour segments.

## Syntax and Arguments

```
session(col_ref, k_integer,unit_of_time) order: order_col [group: group_col]
```

| Argument | Required? | Data Type | Description |
|---|---|---|---|
| col_ref | Y | string | Name of column whose values are applied to the function |
| k_integer | Y | integer (positive) | Length of a session, in combination with `unit_of_time` argument |
| unit_of_time | Y | string | String literal that indicates the units of time to define a session's duration |

For more information on the `order` and `group` parameters, see *Window Transform.*

For more information on syntax standards, see *Language Documentation Syntax Notes.*

**col_ref**

Name of the Datetime column whose values are used to determine sessions.

- Multiple columns and wildcards are not supported.

**Usage Notes:**

| Required? | Data Type | Example Value |
|---|---|---|
| Yes | String (column reference to Datetime values) | `myDates` |

**k_integer**

Defines the length of a session as this number of units, which are defined in the `unit_of_time` parameter.

> **NOTE:** The start of a new session is determined by the first record that is found outside the boundary of the previous session. It is not determined based on any interpretation of a fixed interval.

**Usage Notes:**

| Required? | Data Type | Example Value |
|---|---|---|
| Yes | Integer (positive) | `24` |

**unit_of_time**

Defines the length of each unit of time for purposes of defining the length of a session.

**Usage Notes:**

| Required? | Data Type | Example Value |
|---|---|---|
| Yes | String literal. See below for list. | `hour` |

**Supported values:** `day`, `hour`, `millisecond`, `minute`, `second`

## Examples

> **Tip:** For additional examples, see *Common Tasks.*

### Example - Assign session identifiers to timestamped events

The following dataset contains events from a web site, categorized by customer identifier. Timestamp information is stored in two separate columns, and the imported data is sorted by the `Action` column. Your goal is to generate session identifiers based on sessions of five minute intervals for each customer.

**Source:**

| Date | Time | CustId | Action |
|------|------|--------|--------|
| 2/1/16 | 9:23:00 AM | C001 | change account settings |
| 2/1/16 | 9:23:58 AM | C003 | complete order |
| 2/1/16 | 9:20:00 AM | C002 | login |
| 2/1/16 | 9:20:22 AM | C003 | login |
| 2/1/16 | 9:20:41 AM | C001 | login |
| 2/1/16 | 9:24:52 AM | C004 | login |
| 2/1/16 | 11:24:21 AM | C001 | login |
| 2/1/16 | 9:24:18 AM | C001 | logout |
| 2/1/16 | 9:24:49 AM | C003 | logout |
| 2/1/16 | 9:26:22 AM | C002 | logout |
| 2/1/16 | 9:24:10 AM | C002 | search: bicycles |
| 2/1/16 | 9:23:50 AM | C002 | search: pennyfarthings |
| 2/1/16 | 11:56:09 PM | C004 | search: unicycles |

**Transformation:**

This data makes more sense if it was organized by timestamp of each event. However, the timestamp information is spread across two fields: `Date` and `Time`. Your first step is to consolidate this data into a single field:

| | |
|---|---|
| **Transformation Name** | `Merge columns` |
| **Parameter: Columns** | `Date,Time` |
| **Parameter: Separator** | `' '` |
| **Parameter: New column name** | `'Timestamp'` |

You can now delete the two source columns. After they are deleted, you might notice that the `Timestamp` column is still typed as String data. This typing issue is caused by the AM/PM designators, which you can remove with the following transformation:

| | |
|---|---|
| **Transformation Name** | `Replace text or pattern` |
| **Parameter: Column** | `Timestamp` |
| **Parameter: Find** | `` ` {upper}{2}{end}` `` |
| **Parameter: Replace with** | `''` |

Now that you have valid Datetime data, you can create session identifiers using the following transformation:

| Transformation Name | `Window` |
|---|---|
| Parameter: Formulas | `session(Timestamp, 5, minute)` |
| Parameter: Group by | `CustId` |
| Parameter: Order by | `Timestamp` |

The above transform creates session identifiers from the data in the `Timestamp` column for five-minute intervals. Data is initially grouped by `CustId` and then sorted by `Timestamp` before the `SESSION` function is applied.

You can choose to rename the generated column:

| Transformation Name | `Rename columns` |
|---|---|
| Parameter: Option | `Manual rename` |
| Parameter: Column | `window` |
| Parameter: New column name | `'SessionId'` |

**Results:**

| Timestamp | CustId | Action | SessionId |
|---|---|---|---|
| 2/1/2016 11:24:21 | C001 | login | 1 |
| 2/1/2016 9:20:41 | C001 | login | 2 |
| 2/1/2016 9:23:00 | C001 | change account settings | 2 |
| 2/1/2016 9:24:18 | C001 | logout | 2 |
| 2/1/2016 9:20:22 | C003 | login | 1 |
| 2/1/2016 9:23:58 | C003 | complete order | 1 |
| 2/1/2016 9:24:49 | C003 | logout | 1 |
| 2/1/2016 9:20:00 | C002 | login | 1 |
| 2/1/2016 9:23:50 | C002 | search: pennyfarthings | 1 |
| 2/1/2016 9:24:10 | C002 | search: bicycles | 1 |
| 2/1/2016 9:26:22 | C002 | logout | 1 |
| 1/31/2016 11:56:09 | C004 | search: unicycles | 1 |
| 2/1/2016 9:24:52 | C004 | login | 2 |

Notes:

- Dataset is grouped by `CustId`, but the order of those groupings is determined by the first timestamp for each customer. So, `C003` data appears before `C002`.
- `C001` and `C004` data result in two separate sessions.

# Other Functions

These additional functions provide a variety of useful capabilities for wrangling your data.

# COALESCE Function

Function returns the first non-missing value found in an array of columns.

The order of the columns listed in the function determines the order in which they are searched.

**Wrangle vs. SQL:** This function is part of Wrangle , a proprietary data transformation language. Wrangle is not SQL. For more information, see *Wrangle Language*.

## Basic Usage

```
coalesce([col1,col2,col3])
```

**Output:** Returns the first non-missing detected in `col1`, `col2`, or `col3` in that order.

## Syntax and Arguments

```
coalesce([col_ref1,col_ref2, col_ref3])
```

A reference to a single column does not require brackets. References to multiple columns must be passed to the function as an array of column names.

| Argument | Required? | Data Type | Description |
|----------|-----------|-----------|-------------|
| col_ref1 | Y | string | Name of the first column to find the first non-missing value |
| col_ref2 | N | string | Name of the second column to find the first non-missing value |
| col_ref3 | N | string | Name of the third column to find the first non-missing value |

For more information on syntax standards, see *Language Documentation Syntax Notes*.

**col_ref1, col_ref2, col_ref3**

Name of the column(s) searched for the first non-missing value.

**Usage Notes:**

| Required? | Data Type | Example Value |
|-----------|-----------|---------------|
| Yes | String (column reference) | `[myColumn1, myColumn2]` |

## Examples

> **Tip:** For additional examples, see *Common Tasks*.

**Example - Find first time**

You are tracking multiple racers across multiple heats. Racers might sit out heats for various reasons.

**Source:**

Here's the race data.

| Racer | Heat1 | Heat2 | Heat3 |
|-------|-------|-------|-------|
| Racer X | | 38.22 | 37.61 |
| Racer Y | 41.33 | | 38.04 |
| Racer Z | 39.27 | 39.04 | 38.85 |

**Transformation:**

Use the following transform to grab the first non-missing value from the Heat columns:

| Transformation Name | `New formula` |
|---------------------|-----------|
| **Parameter: Formula type** | `Single row formula` |
| **Parameter: Formula** | `coalesce([Heat1, Heat2, Heat3])` |
| **Parameter: New column name** | `'firstTime'` |

**Results:**

| Racer | Heat1 | Heat2 | Heat3 | firstTime |
|-------|-------|-------|-------|-----------|
| Racer X | | 38.22 | 37.61 | 38.22 |
| Racer Y | 41.33 | | 38.04 | 41.33 |
| Racer Z | 39.27 | 39.04 | 38.85 | 39.27 |

# RAND Function

**Contents:**

---

The `RAND` function generates a random real number between 0 and 1. The function accepts an optional integer parameter, which causes the same set of random numbers to be generated with each job execution.

- This function generates values of Decimal type with fifteen digits of precision after the decimal point. If you want to see all digits in the generated value, you might need to apply a different number format. See *NUMFORMAT Function.*
- New random numbers are generated within the browser, after each browser refresh, and between subsequent job executions.

Optionally, you can insert an integer as a parameter.

- When this value is present, this **seed value** is used as part of the random number generator such that its output is a set of pseudo-random values, which are consistent between job executions.
- When the browser is refreshed, the random numbers remain consistent when the seed value is present.
- This value must be a valid literal Integer value.
- If none is provided, a seed is generated based on the system timestamp.

Column references or functions returning Integer values are not supported.

**Wrangle vs. SQL:** This function is part of Wrangle , a proprietary data transformation language. Wrangle is not SQL. For more information, see *Wrangle Language*.

## Basic Usage

**Example:**

```
rand()
```

**Output:** For each row, generate a random number between 0 and 1 in the new `random` function.

**Example with seed value:**

```
rand(2)
```

**Output:** For each row, generate a random number between 0 and 1 in the new `random` function. The generated random set of random values are consistent between job executions and are, in part, governed by the seed value `2`.

## Syntax and Arguments

There are no arguments for this function.

```
rand([int_value])
```

| Argument | Required? | Data Type | Description |
|---|---|---|---|
| int_value | N | integer | Integer literal |

For more information on syntax standards, see *Language Documentation Syntax Notes*.

**int_value**

Optional Integer literal that is used to generate random numbers. Use of a seed value ensures consistency of output between job executions.

- This value must be a valid literal Integer value.
- Literal numeric values should not be quoted. Quoted values are treated as strings.
- Multiple columns and wildcards are not supported.

**Usage Notes:**

| Required? | Data Type | Example Value |
|---|---|---|
| No | Integer literal | 14 |

## Examples

> **Tip:** For additional examples, see *Common Tasks*.

**Example - Random values**

In the following example, the `random` column is generated by the `RAND` function:

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | rand() |
| Parameter: New column name | 'random' |

| source | random |
|---|---|
| A | 0.516845703365675 |
| B | 0.71118736300207 |
| C | 0.758686066027118 |
| D | 0.640146255791255 |

**Example - Type check functions**

The `RAND` function is typically used to introduce randomness of some kind in your data. In the following example, it is used to perform sampling within your wider dataset.

> **Tip:** Keep in mind that for larger datasets the application displays only a sample of them. This method of randomization is applied to the full dataset during job execution.

**Source:**

You want to extract a random sample of 20% of your set of orders for further study:

| OrderId | Qty | ProdId |
|---------|-----|--------------|
| 1001 | 30 | Widgets |
| 1002 | 10 | Big Widgets |
| 1003 | 5 | Big Widgets |
| 1004 | 40 | Widgets |
| 1005 | 80 | Tiny Widgets |
| 1006 | 20 | Widgets |
| 1007 | 100 | Tiny Widgets |

**Transformation:**

You can use the following transform to generate a random integer from one to 10:

| Transformation Name | `New formula` |
|---------------------|----------------------|
| Parameter: Formula type | `Single row formula` |
| Parameter: Formula | `round(rand() * 10)` |
| Parameter: New column name | `'random'` |

You can now use the following transform to keep only the rows that contain random values that are in the top 20%, where the value is `9` or `10`:

| Transformation Name | `Filter rows` |
|----------------------------|-----------------------|
| Parameter: Condition | `Custom formula` |
| Parameter: Type of formula | `Custom single` |
| Parameter: Condition | `(random > 8)` |
| Parameter: Action | `Keep matching rows` |

**Results:**

> **NOTE:** Since the results are randomized, your results might vary.

| OrderId | Qty | ProdId | random |
|---------|-----|--------------|--------|
| 1005 | 80 | Tiny Widgets | 9 |
| 1007 | 100 | Tiny Widgets | 10 |

# RANDBETWEEN Function

Generates a random integer between a low and a high number. Two inputs may be Integer or Decimal types, functions returning these types, or column references.

- Range is inclusive of the two parameter values.
- The first parameter must be the lower value in the range.

**Wrangle vs. SQL:** This function is part of Wrangle , a proprietary data transformation language. Wrangle is not SQL. For more information, see *Wrangle Language*.

## Basic Usage

```
randbetween(1,10)
```

**Output:** Generates a random Integer value between 1 and `10`.

## Syntax and Arguments

```
<span>RANDBETWEEN(value1,value2)</span>
```

| Argument | Required? | Data Type | Description |
|----------|-----------|-----------|-------------|
| value1 | Y | Integer or Decimal | Integer or Decimal literal, function returning one of these data types, or a column reference for the lower boundary of the range. Range is inclusive of this value. |
| value2 | Y | Integer or Decimal | Integer or Decimal literal, function returning one of these data types, or a column reference for the upper boundary of the range. Range is inclusive of this value. |

For more information on syntax standards, see *Language Documentation Syntax Notes*.

**value1, value2**

Literals, functions, or column references to Integer or Decimal values that are used as the lower and upper bounds, respectively, for the range.

- Missing input values generate missing results.
- Multiple columns and wildcards are not supported.

**Usage Notes:**

| Required? | Data Type | Example Value |
|-----------|-----------|---------------|
| Yes | Integer or Decimal literal, function, or column reference | `100` |

## Examples

> **Tip:** For additional examples, see *Common Tasks*.

**Example - RANDBETWEEN, PI, and TRUNC functions**

This example illustrates how you can apply functions to generate random numeric data in your dataset.

**Functions**:

| Item | Description |
|---|---|
| RANDBETWEEN Function | Generates a random integer between a low and a high number. Two inputs may be Integer or Decimal types, functions returning these types, or column references. |
| PI Function | The `PI` function generates the value of pi to 15 decimal places: 3.1415926535897932. |
| ROUND Function | Rounds input value to the nearest integer. Input can be an Integer, a Decimal, a column reference, or an expression. Optional second argument can be used to specify the number of digits to which to round. |
| TRUNC Function | Removes all digits to the right of the decimal point for any value. Optionally, you can specify the number of digits to which to round. Input can be an Integer, a Decimal, a column reference, or an expression. |
| POW Function | Computes the value of the first argument raised to the value of the second argument. |

**Source:**

In the following example, a company produces 10 circular parts, the size of which is measured in each product's radius in inches.

| prodId | radius_in |
|---|---|
| p001 | 1 |
| p002 | 2 |
| p003 | 3 |
| p004 | 4 |
| p005 | 5 |
| p006 | 6 |
| p007 | 7 |
| p008 | 8 |
| p009 | 9 |
| p010 | 10 |

Based on the above data, the company wants to generate some additional sizing information for these circular parts, including the generation of two points along each part's circumference where quality stress tests can be applied.

**Transformation:**

To begin, you can use the following steps to generate the area and circumference for each product, rounded to three decimal points:

| Transformation Name | `New formula` |
|---|---|
| **Parameter: Formula type** | `Single row formula` |
| **Parameter: Formula** | `ROUND(PI() * (POW(radius_in, 2)), 3)` |
| **Parameter: New column name** | `'area_sqin'` |

| Transformation Name | `New formula` |
|---|---|
| **Parameter: Formula type** | `Single row formula` |

| Parameter: Formula | ROUND(PI() * (2 * radius_in), 3) |
|---|---|
| Parameter: New column name | 'circumference_in' |

For quality purposes, the company needs two tests points along the circumference, which are generated by calculating two separate random locations along the circumference. Since the RANDBETWEEN function only calculates using Integer values, you must first truncate the values from circumference_in:

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | TRUNC(circumference_in) |
| Parameter: New column name | 'trunc_circumference_in' |

Then, you can calculate the random points using the following:

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | RANDBETWEEN(0, trunc_circumference_in) |
| Parameter: New column name | 'testPt01_in' |

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | RANDBETWEEN(0, trunc_circumference_in) |
| Parameter: New column name | 'testPt02_in' |

**Results:**

After the trunc_circumference_in column is dropped, the data should look similar to the following:

| prodId | radius_in | area_sq_in | circumference_in | testPt01_in | testPt02_in |
|---|---|---|---|---|---|
| p001 | 1 | 3.142 | 6.283 | 5 | 5 |
| p002 | 2 | 12.566 | 12.566 | 3 | 3 |
| p003 | 3 | 28.274 | 18.850 | 13 | 13 |
| p004 | 4 | 50.265 | 25.133 | 24 | 24 |
| p005 | 5 | 78.540 | 31.416 | 0 | 0 |
| p006 | 6 | 113.097 | 37.699 | 15 | 15 |
| p007 | 7 | 153.938 | 43.982 | 11 | 11 |
| p008 | 8 | 201.062 | 50.265 | 1 | 1 |
| p009 | 9 | 254.469 | 56.549 | 29 | 29 |
| p010 | 10 | 314.159 | 62.832 | 21 | 21 |

# PI Function

The `PI` function generates the value of pi to 15 decimal places: 3.1415926535897932.

This function uses no parameters. Generated values are of Decimal type and have fifteen digits of precision after the decimal point. If you want to see a fewer number of digits in the generated value, you might need to apply a different number format or using the ROUND function.

- See *NUMFORMAT Function*.
- See *ROUND Function*.

**Wrangle vs. SQL:** This function is part of Wrangle , a proprietary data transformation language. Wrangle is not SQL. For more information, see *Wrangle Language*.

## Basic Usage

```
pi()
```

**Output:** Returns the value of pi.

## Syntax and Arguments

There are no arguments for this function.

## Examples

> **Tip:** For additional examples, see *Common Tasks*.

**Example - pi column**

In the following example, the source is simply the `source` column, and the `pi` column is generated by the `PI` function:

| Transformation Name | New formula |
| --- | --- |
| Parameter: Formula type | Single row formula |
| Parameter: Formula | `pi()` |
| Parameter: New column name | `'pi'` |

| source | pi |
| --- | --- |
| A | 3.1415926535897932 |
| B | 3.1415926535897932 |
| C | 3.1415926535897932 |
| D | 3.1415926535897932 |

### Example - RANDBETWEEN, PI, and TRUNC functions

This example illustrates how you can apply functions to generate random numeric data in your dataset.

**Functions**:

| Item | Description |
|------|-------------|
| RANDBETWE EN Function | Generates a random integer between a low and a high number. Two inputs may be Integer or Decimal types, functions returning these types, or column references. |
| PI Function | The `PI` function generates the value of pi to 15 decimal places: 3.1415926535897932. |
| ROUND Function | Rounds input value to the nearest integer. Input can be an Integer, a Decimal, a column reference, or an expression. Optional second argument can be used to specify the number of digits to which to round. |
| TRUNC Function | Removes all digits to the right of the decimal point for any value. Optionally, you can specify the number of digits to which to round. Input can be an Integer, a Decimal, a column reference, or an expression. |
| POW Function | Computes the value of the first argument raised to the value of the second argument. |

**Source:**

In the following example, a company produces 10 circular parts, the size of which is measured in each product's radius in inches.

| prodId | radius_in |
|--------|-----------|
| p001 | 1 |
| p002 | 2 |
| p003 | 3 |
| p004 | 4 |
| p005 | 5 |
| p006 | 6 |
| p007 | 7 |
| p008 | 8 |
| p009 | 9 |
| p010 | 10 |

Based on the above data, the company wants to generate some additional sizing information for these circular parts, including the generation of two points along each part's circumference where quality stress tests can be applied.

**Transformation:**

To begin, you can use the following steps to generate the area and circumference for each product, rounded to three decimal points:

| Transformation Name | New formula |
|---------------------|-------------|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | ROUND(PI() * (POW(radius_in, 2)), 3) |
| Parameter: New column name | 'area_sqin' |

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | ROUND(PI() * (2 * radius_in), 3) |
| Parameter: New column name | 'circumference_in' |

For quality purposes, the company needs two tests points along the circumference, which are generated by calculating two separate random locations along the circumference. Since the RANDBETWEEN function only calculates using Integer values, you must first truncate the values from circumference_in:

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | TRUNC(circumference_in) |
| Parameter: New column name | 'trunc_circumference_in' |

Then, you can calculate the random points using the following:

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | RANDBETWEEN(0, trunc_circumference_in) |
| Parameter: New column name | 'testPt01_in' |

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | RANDBETWEEN(0, trunc_circumference_in) |
| Parameter: New column name | 'testPt02_in' |

**Results:**

After the trunc_circumference_in column is dropped, the data should look similar to the following:

| prodId | radius_in | area_sq_in | circumference_in | testPt01_in | testPt02_in |
|---|---|---|---|---|---|
| p001 | 1 | 3.142 | 6.283 | 5 | 5 |
| p002 | 2 | 12.566 | 12.566 | 3 | 3 |
| p003 | 3 | 28.274 | 18.850 | 13 | 13 |
| p004 | 4 | 50.265 | 25.133 | 24 | 24 |
| p005 | 5 | 78.540 | 31.416 | 0 | 0 |
| p006 | 6 | 113.097 | 37.699 | 15 | 15 |
| p007 | 7 | 153.938 | 43.982 | 11 | 11 |

| p008 | 8 | 201.062 | 50.265 | 1 | 1 |
| p009 | 9 | 254.469 | 56.549 | 29 | 29 |
| p010 | 10 | 314.159 | 62.832 | 21 | 21 |

# SOURCEROWNUMBER Function

**Contents:**

- *Basic Usage*
- *Syntax and Arguments*
- *Examples*
    - *Example - Header from row that is not the first one*
    - *Example - Using sourcerownumber to create unique row identifiers*
    - *Example - Delete rows based on source row numbers*

---

> **NOTE:** This function has been superseded by the `$sourcerownumber` reference. While this function is still usable in the product, it is likely to be deprecated in a future release. Please use `$sourcerownumber` instead. For more information, see *Source Metadata References*.

Returns the row number of the current row as it appeared in the original source dataset before any steps had been applied.

The following transforms might make original row information invalid or otherwise unavailable. In these cases, the function returns null values:

- `pivot`
- `flatten`
- `join`
- `lookup`
- `union`
- `unnest`
- `unpivot`

> **NOTE:** This function does not apply to relational database sources.

> **NOTE:** If the dataset is sourced from multiple files, a predictable original source row number cannot be guaranteed, and null values are returned.

> **Tip:** If the source row information is still available, you can hover over the left side of a row in the data grid to see the source row number in the original source data.

**Wrangle vs. SQL:** This function is part of Wrangle , a proprietary data transformation language. Wrangle is not SQL. For more information, see *Wrangle Language*.

## Basic Usage

**Example:**

```
sourcerownumber()
```

**Output:** Returns the source row number for each row as it appeared in the original data.

**Sort Example:**

| Transformation Name | Sort rows |
|---|---|
| Parameter: Sort by | sourcerownumber() |

**Output:** Rows in the dataset are re-sorted according to the original order in the dataset.

**Delete Example:**

| Transformation Name | Filter rows |
|---|---|
| Parameter: Condition | Custom formula |
| Parameter: Type of formula | Custom single |
| Parameter: Condition | sourcerownumber() > 101 |
| Parameter: Action | Delete matching rows |

**Output:** Deletes the rows in the dataset that were after row #101 in the original source data.

## Syntax and Arguments

There are no arguments for this function.

## Examples

> **Tip:** For additional examples, see *Common Tasks*.

**Example - Header from row that is not the first one**

This example illustrates how you can rename columns based on the contents of specified rows.

**Source:**

You have imported the following racer data on heat times from a CSV file. When loaded in the Transformer page, it looks like the following:

| (rowId) | column2 | column3 | column4 | column5 |
|---|---|---|---|---|
| 1 | Racer | Heat 1 | Heat 2 | Heat 3 |
| 2 | Racer X | 37.22 | 38.22 | 37.61 |
| 3 | Racer Y | 41.33 | DQ | 38.04 |
| 4 | Racer Z | 39.27 | 39.04 | 38.85 |

In the above, the (rowId) column references the row numbers displayed in the data grid; it is not part of the dataset. This information is available when you hover over the black dot on the left side of the screen.

**Transformation:**

You have examined the best performance in each heat according to the sample. You then notice that the data contains headers, but you forget how it was originally sorted. The data now looks like the following:

| (rowId) | column2 | column3 | column4 | column5 |
|---------|---------|---------|---------|---------|
| 1 | Racer Y | 41.33 | DQ | 38.04 |
| 2 | Racer | Heat 1 | Heat 2 | Heat 3 |
| 3 | Racer X | 37.22 | 38.22 | 37.61 |
| 4 | Racer Z | 39.27 | 39.04 | 38.85 |

You can use the following transformation to use the third row as your header for each column:

| Transformation Name | Rename column with row(s) |
|---------------------|---------------------------|
| Parameter: Option | Use row(s) as column names |
| Parameter: Type | Use a single row to name columns |
| Parameter: Row number | 3 |

**Results:**

After you have applied the above transformation, your data should look like the following:

| (rowId) | Racer | Heat_1 | Heat_2 | Heat_3 |
|---------|-------|--------|--------|--------|
| 3 | Racer Y | 41.33 | DQ | 38.04 |
| 2 | Racer X | 37.22 | 38.22 | 37.61 |
| 4 | Racer Z | 39.27 | 39.04 | 38.85 |

## Example - Using sourcerownumber to create unique row identifiers

The following example demonstrates how to unpack nested data. As part of this example, the SOURCEROWNUMBER function is used as part of a method to create unique row identifiers.

This example illustrates you to use the flatten and unnest transforms.

**Source:**

You have the following data on student test scores. Scores on individual scores are stored in the Scores array, and you need to be able to track each test on a uniquely identifiable row. This example has two goals:

1. One row for each student test
2. Unique identifier for each student-score combination

| LastName | FirstName | Scores |
|----------|-----------|--------|
| Adams | Allen | [81,87,83,79] |
| Burns | Bonnie | [98,94,92,85] |
| Cannon | Charles | [88,81,85,78] |

**Transformation:**

When the data is imported from CSV format, you must add a `header` transform and remove the quotes from the `Scores` column:

| Transformation Name | Rename column with row(s) |
| --- | --- |
| **Parameter: Option** | Use row(s) as column names |
| **Parameter: Type** | Use a single row to name columns |
| **Parameter: Row number** | 1 |

| Transformation Name | Replace text or pattern |
| --- | --- |
| **Parameter: Column** | colScores |
| **Parameter: Find** | '\"' |
| **Parameter: Replace with** | '' |
| **Parameter: Match all occurrences** | true |

**Validate test date:** To begin, you might want to check to see if you have the proper number of test scores for each student. You can use the following transform to calculate the difference between the expected number of elements in the `Scores` array (4) and the actual number:

| Transformation Name | New formula |
| --- | --- |
| **Parameter: Formula type** | Single row formula |
| **Parameter: Formula** | (4 - arraylen(Scores)) |
| **Parameter: New column name** | 'numMissingTests' |

When the transform is previewed, you can see in the sample dataset that all tests are included. You might or might not want to include this column in the final dataset, as you might identify missing tests when the recipe is run at scale.

**Unique row identifier:** The `Scores` array must be broken out into individual rows for each test. However, there is no unique identifier for the row to track individual tests. In theory, you could use the combination of `LastName-FirstName-Scores` values to do so, but if a student recorded the same score twice, your dataset has duplicate rows. In the following transform, you create a parallel array called `Tests`, which contains an index array for the number of values in the `Scores` column. Index values start at `0`:

| Transformation Name | New formula |
| --- | --- |
| **Parameter: Formula type** | Single row formula |
| **Parameter: Formula** | range(0,arraylen(Scores)) |
| **Parameter: New column name** | 'Tests' |

Also, we will want to create an identifier for the source row using the `sourcerownumber` function:

| Transformation Name | New formula |
| --- | --- |

| Parameter: Formula type | Single row formula |
|---|---|
| Parameter: Formula | sourcerownumber() |
| Parameter: New column name | 'orderIndex' |

**One row for each student test:** Your data should look like the following:

| LastName | FirstName | Scores | Tests | orderIndex |
|---|---|---|---|---|
| Adams | Allen | [81,87,83,79] | [0,1,2,3] | 2 |
| Burns | Bonnie | [98,94,92,85] | [0,1,2,3] | 3 |
| Cannon | Charles | [88,81,85,78] | [0,1,2,3] | 4 |

Now, you want to bring together the `Tests` and `Scores` arrays into a single nested array using the `arrayzip` function:

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | arrayzip([Tests,Scores]) |

Your dataset has been changed:

| LastName | FirstName | Scores | Tests | orderIndex | column1 |
|---|---|---|---|---|---|
| Adams | Allen | [81,87,83,79] | [0,1,2,3] | 2 | [[0,81],[1,87],[2,83],[3,79]] |
| Adams | Bonnie | [98,94,92,85] | [0,1,2,3] | 3 | [[0,98],[1,94],[2,92],[3,85]] |
| Cannon | Charles | [88,81,85,78] | [0,1,2,3] | 4 | [[0,88],[1,81],[2,85],[3,78]] |

Use the following to unpack the nested array:

| Transformation Name | Expand arrays to rows |
|---|---|
| Parameter: Column | column1 |

Each test-score combination is now broken out into a separate row. The nested Test-Score combinations must be broken out into separate columns using the following:

| Transformation Name | Unnest Objects into columns |
|---|---|
| Parameter: Column | column1 |
| Parameter: Paths to elements | '[0]','[1]' |

After you delete `column1`, which is no longer needed you should rename the two generated columns:

| Transformation Name | Rename columns |
|---|---|
| Parameter: Option | Manual rename |
| Parameter: Column | column_0 |
| Parameter: New column | 'TestNum' |

| Transformation Name | Rename columns |
|---|---|
| Parameter: Option | `Manual rename` |
| Parameter: Column | `column_1` |
| Parameter: New column name | `'TestScore'` |

**Unique row identifier:** You can do one more step to create unique test identifiers, which identify the specific test for each student. The following uses the original row identifier `OrderIndex` as an identifier for the student and the `TestNumber` value to create the `TestId` column value:

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | `Single row formula` |
| Parameter: Formula | `(orderIndex * 10) + TestNum` |
| Parameter: New column name | `'TestId'` |

The above are integer values. To make your identifiers look prettier, you might add the following:

| Transformation Name | Merge columns |
|---|---|
| Parameter: Columns | `'TestId00','TestId'` |

**Extending:** You might want to generate some summary statistical information on this dataset. For example, you might be interested in calculating each student's average test score. This step requires figuring out how to properly group the test values. In this case, you cannot group by the `LastName` value, and when executed at scale, there might be collisions between first names when this recipe is run at scale. So, you might need to create a kind of primary key using the following:

| Transformation Name | Merge columns |
|---|---|
| Parameter: Columns | `'LastName','FirstName'` |
| Parameter: Separator | `'-'` |
| Parameter: New column name | `'studentId'` |

You can now use this as a grouping parameter for your calculation:

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | `Single row formula` |
| Parameter: Formula | `average(TestScore)` |
| Parameter: Group rows by | `studentId` |
| Parameter: New column name | `'avg_TestScore'` |

**Results:**

After you delete unnecessary columns and move your columns around, the dataset should look like the following:

| TestId | LastName | FirstName | TestNum | TestScore | studentId | avg_TestScore |
|--------|----------|-----------|---------|-----------|-----------|---------------|
| TestId0021 | Adams | Allen | 0 | 81 | Adams-Allen | 82.5 |
| TestId0022 | Adams | Allen | 1 | 87 | Adams-Allen | 82.5 |
| TestId0023 | Adams | Allen | 2 | 83 | Adams-Allen | 82.5 |
| TestId0024 | Adams | Allen | 3 | 79 | Adams-Allen | 82.5 |
| TestId0031 | Adams | Bonnie | 0 | 98 | Adams-Bonnie | 92.25 |
| TestId0032 | Adams | Bonnie | 1 | 94 | Adams-Bonnie | 92.25 |
| TestId0033 | Adams | Bonnie | 2 | 92 | Adams-Bonnie | 92.25 |
| TestId0034 | Adams | Bonnie | 3 | 85 | Adams-Bonnie | 92.25 |
| TestId0041 | Cannon | Chris | 0 | 88 | Cannon-Chris | 83 |
| TestId0042 | Cannon | Chris | 1 | 81 | Cannon-Chris | 83 |
| TestId0043 | Cannon | Chris | 2 | 85 | Cannon-Chris | 83 |
| TestId0044 | Cannon | Chris | 3 | 78 | Cannon-Chris | 83 |

**Example - Delete rows based on source row numbers**

**Source:**

Your dataset is the following set of orders.

| CustId | FirstName | LastName | City | State | LastOrder |
|--------|-----------|----------|------|-------|-----------|
| 1001 | Skip | Jones | San Francisco | CA | 25 |
| 1002 | Adam | Allen | Oakland | CA | 1099 |
| 1003 | David | Wiggins | Oakland | MI | 125.25 |
| 1004 | Amanda | Green | Detroit | MI | 452.5 |
| 1005 | Colonel | Mustard | Los Angeles | CA | 950 |
| 1006 | Pauline | Hall | Sagninaw | MI | 432.22 |
| 1007 | Sarah | Miller | Cheyenne | WY | 724.22 |
| 1008 | Teddy | Smith | Juneau | AK | 852.11 |
| 1009 | Joelle | Higgins | Sacramento | CA | 100 |

**Transformation:**

Initially, you want to review your list of orders by last name.

| Transformation Name | Sort rows |
|---------------------|-----------|
| Parameter: Sort by | LastName |

During your review, you notice that two customer orders are no longer valid and need to be removed. They are:

- LastName: Hall
- LastName: Jones

You might hover over the left side of the screen to reveal the row numbers. You select the row numbers for each of these rows, and a delete suggestion is provided for you. When you click **Modify**, you see the following transformation:

| Transformation Name | `Filter rows` |
|---|---|
| **Parameter: Condition** | `Custom formula` |
| **Parameter: Type of formula** | `Custom single` |
| **Parameter: Condition** | `in(sourcerownumber(), [2,7])` |
| **Parameter: Action** | `Delete matching rows` |

The above checks the results of the `sourcerownumber` function, which returns the original row order for the selected rows. If a selected row matches values in the `[2,7]` array of row numbers, then the row is deleted.

**Results:**

When the preceding transform is added, your dataset looks like the following, and your sort order is maintained:

**Source:**

| CustId | FirstName | LastName | City | State | LastOrder |
|---|---|---|---|---|---|
| 1002 | Adam | Allen | Oakland | CA | 1099 |
| 1004 | Amanda | Green | Detroit | MI | 452.5 |
| 1009 | Joelle | Higgins | Sacramento | CA | 100 |
| 1007 | Sarah | Miller | Cheyenne | WY | 724.22 |
| 1005 | Colonel | Mustard | Los Angeles | CA | 950 |
| 1008 | Teddy | Smith | Juneau | AK | 852.11 |
| 1003 | David | Wiggins | Oakland | MI | 125.25 |

# IF Function

**Contents:**

The `IF` function allows you to build if/then/else conditional logic within your transforms.

> **NOTE:** The `IF` function is interchangeable with ternary operators. You should use this function instead of ternary construction.

> **NOTE:** If you are running your job on Spark, avoid creating single conditional transformations with deeply nested sets of conditions. On Spark, these jobs can time out, and deeply nested steps can be difficult to debug. Instead, break up your nesting into smaller conditional transformations of multiple steps.

**Wrangle vs. SQL:** This function is part of Wrangle , a proprietary data transformation language. Wrangle is not SQL. For more information, see *Wrangle Language*.

## Basic Usage

**Example:**

```
if(State == &apos;NY&apos;,&apos;New York, New York!&apos;,&apos;some other place&apos;)
```

**Output:** If the value in the `State` column is `NY`, return the value `New York, New York!`. Otherwise, the returned valueis `some other place`.

**Nested IF Example:**

You can build `IF` statements within `IF` statements as in the following example, in which the second `IF` is evaluated if the first one evaluates to `false`:

```
if(State == &apos;NY&apos;,0.05,if(State==&apos;CA&apos;,0.08,0))
```

A more detailed nested example is available below.

## Syntax and Arguments

In the following, if the `test expression` evaluates to `true`, the `true_expression` is executed. Otherwise, the `false_expression` is executed.

```
if(test_expression, true_expression,false_expression)
```

| Argument | Required? | Data Type | Description |
|---|---|---|---|
| test_expression | Y | string | Expression that is evaluated. Must resolve to `true` or `false` |
| true_expression | Y | string | Expression that is executed if `test_expression` is `true` |
| false_expression | N | string | Expression that is executed if `test_expression` is `false` |

All of these expressions can be constants (strings, integers, or any other supported literal value) or sophisticated elements of logic, although the test expression must evaluate to a Boolean value.

For more information on syntax standards, see *Language Documentation Syntax Notes*.

### test_expression

This parameter contains the expression to evaluate. This expression must resolve to a Boolean (`true` or `false`) value.

**Usage Notes:**

| Required? | Data Type | Example Value |
|---|---|---|
| Yes | String (expression that evaluates to `true` or `false` | `(LastName == 'Mouse' && FirstName == 'Mickey')` |

### true_expression, false_expression

The `true_expression` determines the value or conditional that is generated if the `test_expression` evaluates to `true`. If the test is `false`, then the `false_expression` applies.

These expressions typically generate output values and can use a combination of literals, functions, and column references.

- A true expression is required. You can insert a blank expression ( `""`).
- If a false expression is not provided, false results yield a value of `false`.

**Usage Notes:**

| Required? | Data Type | Example Value |
|---|---|---|
| Yes | String (expression) | See examples below. |

## Examples

> **Tip:** For additional examples, see *Common Tasks*.

**Example - Basic Usage**

Example data:

| X | Y |
|---|---|
| true | true |
| true | false |
| false | true |

| false | false |

**Transformation:**

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | if((X == Y), 'yes','no') |
| Parameter: New column name | 'equals' |

**Results:**

Your output looks like the following:

| X | Y | equals |
|---|---|---|
| true | true | yes |
| true | false | no |
| false | true | no |
| false | false | yes |

### Example - Stock Quotes

This example demonstrates how you can chain together multiple if/then/else conditions within a single transform step.

You have a set of stock prices that you want to analyze. Based on a set of rules, you want to determine any buy, sell, or hold action to take.

**Source:**

| Ticket | Qty | BuyPrice | CurrentPrice |
|---|---|---|---|
| GOOG | 10 | 705.25 | 674.5 |
| FB | 100 | 84.00 | 101.125 |
| AAPL | 50 | 125.25 | 97.375 |
| MSFT | 100 | 38.875 | 45.25 |

**Transformation:**

You can perform evaluations of this data using the IF function to determine if you want to take action.

To assist in evaluation, you might first want to create columns that contain the cost (Basis) and the current value (CurrentValue) for each stock:

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | (Qty * BuyPrice) |
| Parameter: New column name | 'Basis' |

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | (Qty * CurrentPrice) |
| Parameter: New column name | 'CurrentValue' |

Now, you can build some rules based on the spread between `Basis` and `CurrentValue`.

**Single `IF` version:** In this case, the most important action is determining if it is time to sell. The following rule writes a `sell` notification if the current value is $1000 or more than the cost. Otherwise, no value is written to the action column.

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | if((CurrentValue - 1000 > Basis), 'sell','') |
| Parameter: New column name | 'action' |

**Nested `IF` version:** But what about buying more? The following transform is an edit to the previous one. In this new version, the sell test is performed, and if `false`, writes a `buy` action if the `CurrentPrice` is within 10% of the `BuyPrice`.

This second evaluation is performed after the first one, as it replaces the else clause, which did nothing in the previous version. In the Recipe Panel, click the previous transform and edit it, replacing it with the new version:

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | if((CurrentValue - 1000 > Basis), 'sell', if((abs(CurrentValue - Basis) <= (Basis * 0.1)),'buy','hold')) |
| Parameter: New column name | 'action' |

If neither test evaluates to `true`, the written action is `hold`.

You might want to format some of your columns using dollar formatting, as in the following:

> **NOTE:** The following formatting inserts a dollar sign ($) in front of the value, which changes the data type to String.

| Transformation Name | Edit column with formula |
|---|---|
| Parameter: Columns | BuyPrice |
| Parameter: Formula | numformat(BuyPrice, '$ ##,###.00') |

**Results:**

After moving your columns, your dataset should look like the following, if you completed the number formatting steps:

| Ticket | Qty | BuyPrice | CurrentPrice | Basis | CurrentValue | action |
|---|---|---|---|---|---|---|
| GOOG | 10 | 705.25 | $ 674.50 | $ 7,052.50 | $ 6,745.00 | buy |
| FB | 100 | 84.00 | $ 101.13 | $ 8,400.00 | $ 10,112.50 | sell |
| AAPL | 50 | 125.25 | $ 97.38 | $ 6,262.50 | $ 4,868.75 | hold |
| MSFT | 100 | 38.88 | $ 45.25 | $ 3,887.50 | $ 4,525.00 | hold |

# CASE Function

**Contents:**

The `CASE` function allows you to perform multiple conditional tests on a set of expressions within a single statement. When a test evaluates to `true`, a corresponding output is generated. Outputs may be a literal or expression.

For more information on the `IF` function, see *IF Function*.

**Wrangle vs. SQL:** This function is part of Wrangle , a proprietary data transformation language. Wrangle is not SQL. For more information, see *Wrangle Language*.

## Basic Usage

**Example:**

```
case([ Qty &lt;= 10, &apos;low_qty&apos;, Qty &gt;=100, &apos;high_qty&apos;, &apos;
med_qty&apos;])
```

**Output:** Returns a text string based on the evaluation of the `Qty` column:

- Qty <= 10 : `low_qty`
- Qty >= 100 : `high_qty`
- All other values of Qty : `med_qty`

## Syntax and Arguments

In the following, If the `testX` expression evaluates to `true`, then the `resultX` value is the output.

- Test expressions are evaluated in the listed order.
- Text expressions and results are paired values in an array.
- You must include one or more test expressions.
- Each test must include a result expression. Result expression can be a literal value or an expression that evaluates to a value of a supported data type.
- If a quoted value is included as a test expression, it is evaluated as the value to write for all values that have not yet matched a test (else expression).

```
case([test1, &apos;result1&apos;,test2, &apos;result2&apos;, testn, &apos;resultn&apos;,
&apos;result_else&apos;])
```

| Argument | Required? | Data Type | Description |
|---|---|---|---|
| test1, test2, testn | Y | expression | Expression that is evaluated. Must resolve to `true` or `false` |

| result1, result2, result2, result_else | Y | string | Quoted string that is written if the corresponding test expression evaluates to `true`. |

All of these expressions can be constants (strings, integers, or any other supported literal values) or sophisticated elements of logic, although the test expression must evaluate to a Boolean value.

For more information on syntax standards, see *Language Documentation Syntax Notes*.

### test1, test2, testn

These parameters contain the expressions to evaluate. This expression must resolve to a Boolean (`true` or `false`) value.

> **NOTE:** The syntax of a test expression follows the same syntax as the `IF` function. For example, you must use the double-equals signs to compare values (`status == 'Ok'`).

**Usage Notes:**

| Required? | Data Type | Example Value |
|---|---|---|
| Yes | Expression that evaluates to `true` or `false` | `(OrderAge > 90)` |

### result1, result2, result2, result_else

If the corresponding test expression evaluates to `true`, this value is written as the result.

These expressions can literals of any data type or expressions that evaluate to literals of any data type.

**Usage Notes:**

| Required? | Data Type | Example Value |
|---|---|---|
| Yes | Literal value or expression | See examples below. |

## Examples

> **Tip:** For additional examples, see *Common Tasks*.

### Example - Basic Usage

The following data represents orders received during the week. Discounts are applied to the orders based on the following rules:

- The standard discount is 5%.
- If an order is for fewer less than 10 units, then the discount is reduced by 5%.
- If an order is for more than 20 units, then the discount is increased by 5%.
- The special Friday discount is 2% more than the standard discount.

| OrdDate | CustId | Qty | Std_Disc |
|---|---|---|---|
| 5/8/17 | C001 | 4 | 0.05 |
| 5/9/17 | C002 | 11 | 0.05 |

| 5/10/17 | C003 | 4 | 0.05 |
|---------|------|---|------|
| 5/11/17 | C001 | 25 | 0.05 |
| 5/12/17 | C002 | 19 | 0.05 |

**Transforms:**

To determine the day of the week, you can use the following transformation:

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | weekday(Date) |

You can build the discount rules into the following transform, which generates the `Disc` column:

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | case([Qty<10, Std_Disc - 0.05, Qty>=20, Std_Disc + 0.05, weekday_Date == 5, Std_Disc + 0.02, Std_Disc]) |
| Parameter: New column name | 'Disc' |

**Results:**

| OrdDate | CustId | Qty | Std_Disc | Disc |
|---------|--------|-----|----------|------|
| 5/8/17 | C001 | 4 | 0.05 | 0 |
| 5/9/17 | C002 | 11 | 0.05 | 0.05 |
| 5/10/17 | C003 | 4 | 0.05 | 0 |
| 5/11/17 | C001 | 25 | 0.05 | 0.1 |
| 5/12/17 | C002 | 19 | 0.05 | 0.07 |

# Ternary Operators

Ternary operators allow you to build if/then/else conditional logic within your transforms. Please use the `IF` function instead.

> **NOTE:** Ternary operators have been superseded by the `IF` function. See *IF Function*.

In the following, if the `test expression` evaluates to `true`, the `true_expression` is executed. Otherwise, the `false_expression` is executed.

```
(test_expression) ? (true_expression) : (false_expression)
```

All of these expressions can be constants (strings, integers, or any other supported literal value) or sophisticated elements of logical, although the test expression must evaluate to a Boolean value.

> **NOTE:** If you are running your job on Spark, avoid creating single conditional transformations with deeply nested sets of conditions. On Spark, these jobs can time out, and deeply nested steps can be difficult to debug. Instead, break up your nesting into smaller conditional transformations of multiple steps.

## Usage

Example data:

| X | Y |
|---|---|
| true | true |
| true | false |
| false | true |
| false | false |

**Transforms:**

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | (X == Y) ? 'yes' : 'no' |
| Parameter: New column name | 'equals' |

**Results:**

Your output looks like the following:

| X | Y | equals |
|---|---|---|
| true | true | yes |
| true | false | no |
| false | true | no |

| false | false | yes |
|-------|-------|-----|

## Examples

> **Tip:** For additional examples, see *Common Tasks*.

**Example - Stock Quotes**

You have a set of stock prices that you want to analyze. Based on a set of rules, you want to determine any buy, sell, or hold action to take.

**Source:**

| Ticket | Qty | BuyPrice | CurrentPrice |
|--------|-----|----------|--------------|
| GOOG   | 10  | 705.25   | 674.5        |
| FB     | 100 | 84.00    | 101.125      |
| AAPL   | 50  | 125.25   | 97.375       |
| MSFT   | 100 | 38.875   | 45.25        |

**Transformation:**

You can perform evaluations of this data using ternary operators to determine if you want to take action.

To assist in evaluation, you might first want to create columns that contain the cost (`Basis`) and the current value (`CurrentValue`) for each stock:

| Transformation Name | New formula |
|---------------------|-------------|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | (Qty * BuyPrice) |
| Parameter: New column name | 'Basis' |

| Transformation Name | New formula |
|---------------------|-------------|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | (Qty * CurrentPrice) |
| Parameter: New column name | 'CurrentValue' |

Now, you can build some rules based on the spread between `Basis` and `CurrentValue`.

The most important action is determining if it is time to sell. The following rule writes a `sell` notification if the current value is $1000 or more than the cost. Otherwise, no value is written to the action column.

| Transformation Name | New formula |
|---------------------|-------------|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | (CurrentValue - 1000 > Basis) ? 'sell' : '' |

| Parameter: New column name | 'action' |
|---|---|

But what about buying more? The following transform is an edit to the previous one. In this new version, the sell test is performed, and if writes a `buy` action if the `CurrentPrice` is within 10% of the `BuyPrice`.

This second evaluation is performed after the first one, as it replaces the else clause, which did nothing in the previous version. In the Recipe panel, click the previous transform and edit it, replacing it with the new version:

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | ((CurrentValue - 1000) > Basis) ? 'sell' : ((abs(CurrentValue - Basis) <= (Basis * 0.1)) ? 'buy' : 'hold') |
| Parameter: New column name | 'action' |

If neither test evaluates to `true`, the written action is `hold`.

You might want to format some of your columns using dollar formatting, as in the following:

> **NOTE:** The following formatting inserts a dollar sign ($) in front of the value, which changes the data type to String.

| Transformation Name | Edit column with formula |
|---|---|
| Parameter: Columns | BuyPrice |
| Parameter: Formula | NUMFORMAT(BuyPrice, '$ ##,###.00') |

**Results:**

After moving your columns, your dataset should look like the following, if you completed the number formatting steps:

| Ticket | Qty | BuyPrice | CurrentPrice | Basis | CurrentValue | action |
|---|---|---|---|---|---|---|
| GOOG | 10 | 705.25 | $ 674.50 | $ 7,052.50 | $ 6,745.00 | buy |
| FB | 100 | 84.00 | $ 101.13 | $ 8,400.00 | $ 10,112.50 | sell |
| AAPL | 50 | 125.25 | $ 97.38 | $ 6,262.50 | $ 4,868.75 | hold |
| MSFT | 100 | 38.88 | $ 45.25 | $ 3,887.50 | $ 4,525.00 | hold |

# IPTOINT Function

Computes an integer value for a four-octet internet protocol (IP) address. Source value must be a valid IP address or a column reference to IP addresses.

IP addresses must be in the following format:

```
aaa.bbb.ccc.ddd
```

where `aaa`, `bbb`, `ccc`, and `ddd`, are integers 0 - 255, inclusive.

> **NOTE:** IPv6 addresses are not supported.

The formula used to compute the integer equivalent of the above IP address is the following:

$$(aaa * 256^3) + (bbb * 256^2) + (ccc * 256) + (ddd)$$

As a result, each valid IP address has a unique integer equivalent.

**Wrangle vs. SQL:** This function is part of Wrangle , a proprietary data transformation language. Wrangle is not SQL. For more information, see *Wrangle Language*.

## Basic Usage

**Numeric literal example:**

```
iptoint(&apos;1.2.3.4&apos; )
```

**Output:** Returns the integer value 16909060.

**Column reference example:**

```
iptoint(IpAddr)
```

**Output:** Returns the value of the `IpAddr` column converted to an integer value.

## Syntax and Arguments

```
iptoint(column_ipaddr)
```

| Argument | Required? | Data Type | Description |
|---|---|---|---|
| column_ipaddr | Y | string | Column name or string literal identifying the IP address to convert to an integer value |

For more information on syntax standards, see *Language Documentation Syntax Notes*.

**column_ipaddr**

Name of the column or IP address literal whose values are used to compute the equivalent integer value.

- Missing input values generate missing results.
- Multiple columns and wildcards are not supported.

**Usage Notes:**

| Required? | Data Type | Example Value |
|-----------|-----------|---------------|
| Yes | String literal or column reference (IP address) | `4.3.2.1` |

## Examples

> **Tip:** For additional examples, see *Common Tasks*.

### Example - Convert IP addresses to integers

This examples illustrates how you can convert IP addresses to numeric values for purposes of comparison and sorting.

**Functions**:

| Item | Description |
|------|-------------|
| IPTOINT Function | Computes an integer value for a four-octet internet protocol (IP) address. Source value must be a valid IP address or a column reference to IP addresses. |
| IPFROMINT Function | Computes a four-octet internet protocol (IP) address from a 32-bit integer input. |

**Source:**

Your dataset includes the following values for IP addresses:

| IpAddr |
|--------|
| 192.0.0.1 |
| 10.10.10.10 |
| 1.2.3.4 |
| 1.2.3 |
| http://12.13.14.15 |
| https://16.17.18.19 |

**Transformation:**

When the above data is imported, the application initially types the column as URL values, due to the presence of the `http://` and `https://` protocol identifiers. Select the IP Address data type for the column. The last three values are listed as mismatched values. You can fix the issues with the last two entries by applying the following transform, which matches on both `http://` and `https://` strings:

| Transformation Name | `Replace text or pattern` |
|---------------------|----------------------------|
| **Parameter: Column** | `IpAddr` |
| **Parameter: Find** | `` `http%?://` `` |
| **Parameter: Replace with** | `''` |

Now, only the `1.2.3` value is mismatched. Perhaps you know that there is a missing zero at the end of it. To add it back, you can do the following:

| Transformation Name | `Replace text or pattern` |
| --- | --- |
| **Parameter: Column** | `IpAddr` |
| **Parameter: Find** | `` `1.2.3[end]` `` |
| **Parameter: Replace with** | `'1.2.3.0'` |
| **Parameter: Match all occurrences** | `true` |

All values in the column should be valid for the IP Address data type. To convert these values to their integer equivalents:

| Transformation Name | `New formula` |
| --- | --- |
| **Parameter: Formula type** | `Single row formula` |
| **Parameter: Formula** | `IPTOINT(IpAddr)` |
| **Parameter: New column name** | `'ip_as_int'` |

You can now manipulate the data based on this numeric key. To convert the integer values back to IP addresses for checking purposes, use the following:

| Transformation Name | `New formula` |
| --- | --- |
| **Parameter: Formula type** | `Single row formula` |
| **Parameter: Formula** | `IPFROMINT(ip_as_int)` |
| **Parameter: New column name** | `'ip_check'` |

**Results:**

| X | ip_as_int | ip_check |
| --- | --- | --- |
| 192.0.0.1 | 3221225473 | 192.0.0.1 |
| 10.10.10.10 | 168430090 | 10.10.10.10 |
| 1.2.3.4 | 16909060 | 1.2.3.4 |
| 1.2.3.0 | 16909056 | 1.2.3.0 |
| 12.13.14.15 | 202182159 | 12.13.14.15 |
| 16.17.18.19 | 269554195 | 16.17.18.19 |

# IPFROMINT Function

Computes a four-octet internet protocol (IP) address from a 32-bit integer input.

Source value must be a valid integer within the range specified by the formula below. A valid IPv4 address is in the following format:

```
aaa.bbb.ccc.ddd
```

> **NOTE:** IPv6 addresses are not supported.

The formula used to compute the integer equivalent of an IP address is the following:

$$(aaa * 256^3) + (bbb * 256^2) + (ccc * 256) + (ddd)$$

So, the formula to compute this IP address is the following:

| Input | aaa | bbb | ccc | ddd |
|---|---|---|---|---|
| X | aaa = floor(Input / (256$^3$)) <br><br> remainderA = Input - aaa | bbb = floor(remainderA / (256$^2$)) <br><br> remainderB = remainderA - bbb | ccc = floor(remainderB / 256) <br><br> remainderC = remainderB - ccc | ddd = remainderC |

Output value:

```
Output = aaa + '.' + bbb + '.' + ccc + '.' + ddd
```

**Wrangle vs. SQL:** This function is part of Wrangle , a proprietary data transformation language. Wrangle is not SQL. For more information, see *Wrangle Language*.

## Basic Usage

**Numeric literal example:**

```
ipfromint(&apos;16909060&apos;)
```

**Output:** Returns the IP address `1.2.3.4`.

**Column reference example:**

```
ipfromint(IpInt)
```

**Output:** Returns the values of the `IpInt` column converted to an IP address value.

## Syntax and Arguments

```
ipfromint(column_int)
```

| Argument | Required? | Data Type | Description |
|---|---|---|---|
| column_int | Y | string or integer | Name of column or integer literal that is to be converted to an IP address value |

For more information on syntax standards, see *Language Documentation Syntax Notes*.

**column_int**

Name of the column or integer literal whose values are used to compute the equivalent IP address value.

- Missing input values generate missing results.
- Multiple columns and wildcards are not supported.

**Usage Notes:**

| Required? | Data Type | Example Value |
|-----------|-----------|---------------|
| Yes | Integer literal or column reference | `16909060` |

## Examples

> **Tip:** For additional examples, see *Common Tasks*.

**Example - Convert IP addresses to integers**

This examples illustrates how you can convert IP addresses to numeric values for purposes of comparison and sorting.

**Functions**:

| Item | Description |
|------|-------------|
| IPTOINT Function | Computes an integer value for a four-octet internet protocol (IP) address. Source value must be a valid IP address or a column reference to IP addresses. |
| IPFROMINT Function | Computes a four-octet internet protocol (IP) address from a 32-bit integer input. |

**Source:**

Your dataset includes the following values for IP addresses:

| IpAddr |
|--------|
| 192.0.0.1 |
| 10.10.10.10 |
| 1.2.3.4 |
| 1.2.3 |
| http://12.13.14.15 |
| https://16.17.18.19 |

**Transformation:**

When the above data is imported, the application initially types the column as URL values, due to the presence of the `http://` and `https://` protocol identifiers. Select the IP Address data type for the column. The last three

values are listed as mismatched values. You can fix the issues with the last two entries by applying the following transform, which matches on both `http://` and `https://` strings:

| Transformation Name | `Replace text or pattern` |
|---|---|
| Parameter: Column | `IpAddr` |
| Parameter: Find | `` `http%?://` `` |
| Parameter: Replace with | `''` |

> **NOTE:** The `%?` Pattern matches zero or one time on any character, which enables the matching on both variants of the protocol identifier.

Now, only the `1.2.3` value is mismatched. Perhaps you know that there is a missing zero at the end of it. To add it back, you can do the following:

| Transformation Name | `Replace text or pattern` |
|---|---|
| Parameter: Column | `IpAddr` |
| Parameter: Find | `` `1.2.3[end]` `` |
| Parameter: Replace with | `'1.2.3.0'` |
| Parameter: Match all occurrences | `true` |

All values in the column should be valid for the IP Address data type. To convert these values to their integer equivalents:

| Transformation Name | `New formula` |
|---|---|
| Parameter: Formula type | `Single row formula` |
| Parameter: Formula | `IPTOINT(IpAddr)` |
| Parameter: New column name | `'ip_as_int'` |

You can now manipulate the data based on this numeric key. To convert the integer values back to IP addresses for checking purposes, use the following:

| Transformation Name | `New formula` |
|---|---|
| Parameter: Formula type | `Single row formula` |
| Parameter: Formula | `IPFROMINT(ip_as_int)` |
| Parameter: New column name | `'ip_check'` |

**Results:**

| X | ip_as_int | ip_check |
|---|---|---|
| 192.0.0.1 | 3221225473 | 192.0.0.1 |
| 10.10.10.10 | 168430090 | 10.10.10.10 |

| | | |
|---|---|---|
| 1.2.3.4 | 16909060 | 1.2.3.4 |
| 1.2.3.0 | 16909056 | 1.2.3.0 |
| 12.13.14.15 | 202182159 | 12.13.14.15 |
| 16.17.18.19 | 269554195 | 16.17.18.19 |

# RANGE Function

**Contents:**

Computes an array of integers, from a beginning integer to an end (stop) integer, stepping by a third parameter.

> **NOTE:** If the function generates more than 100,000 values for a cell, the output is a null value.

**Wrangle vs. SQL:** This function is part of Wrangle , a proprietary data transformation language. Wrangle is not SQL. For more information, see *Wrangle Language*.

## Basic Usage

**Numeric literal example:**

```
range(0,3,1 )
```

**Output:** Returns the following array:

```
[0,1,2]
```

**Column reference example:**

```
range(0,MaxValue,stepValue)
```

**Output:** Returns an array of values from zero to the value in the `MaxValue` column stepping by the `stepValue` column value.

## Syntax and Arguments

```
range(column_integer_start, column _integer_end, column_integer_step)
```

| Argument | Required? | Data Type | Description |
|---|---|---|---|
| column_integer_start | Y | string or integer | Name of column or Integer literal that represents the start of the range |
| column_integer_end | Y | string or integer | Name of column or Integer literal that represents the end of the range |
| column_integer_step | Y | string or integer | Name of column or Integer literal that represents the steps in integers between values in the range |

For more information on syntax standards, see *Language Documentation Syntax Notes*.

### column_integer_start

Name of the column or value of the starting integer used to compute the range.

> **NOTE:** This value is always included in the range, unless it is equal to the value for `col-integer-stop`, which results in a blank array.

- Missing input values generate missing results.
- Multiple columns and wildcards are not supported.

**Usage Notes:**

| Required? | Data Type | Example Value |
|-----------|-----------|---------------|
| Yes | Integer | 0 |

### column_integer_end

Name of the column or value of the end integer used to compute the range.

> **NOTE:** This value is not included in the output.

- Missing input values generate missing results.
- Multiple columns and wildcards are not supported.

**Usage Notes:**

| Required? | Data Type | Example Value |
|-----------|-----------|---------------|
| Yes | Integer | 20 |

### column_integer_step

Name of the column or value of the integer used to compute the integer interval (step) between each value in the range.

> **NOTE:** This value must be a positive integer. If `col-integer-start` is greater than `col-integer-stop`, steps are negative values of this parameter.

- Missing input values generate missing results.
- Multiple columns and wildcards are not supported.

**Usage Notes:**

| Required? | Data Type | Example Value |
|-----------|-----------|---------------|
| Yes | Integer | 2 |

# Examples

### Example - Breaking out log messages

### Source:

Your dataset contains log data that is gathered each minute, yet each entry can contain multiple error messages in an array. The key fields might look like the following:

| Timestamp | Errors |
|---|---|
| 02/16/16 15:31 | ["Unable to connect","File not found","Proxy down","conn. timeout"] |
| 02/16/16 15:30 | [] |
| 02/16/16 15:29 | ["Access forbidden","Invalid password"] |

### Transformation:

You can use the following steps to break out the array values into separate rows. The following transform generates a column containing the number of elements in each row's `Errors` array.

| | |
|---|---|
| **Transformation Name** | `New formula` |
| **Parameter: Formula type** | `Single row formula` |
| **Parameter: Formula** | `arraylen(Errors)` |
| **Parameter: New column name** | `'arraylength_Errors'` |

This transform deletes rows that contain no errors:

| | |
|---|---|
| **Transformation Name** | `Filter rows` |
| **Parameter: Condition** | `Custom formula` |
| **Parameter: Type of formula** | `Custom single` |
| **Parameter: Condition** | `(arraylength_Errors == 0)` |
| **Parameter: Action** | `Delete matching rows` |

For the remaining rows, you can generate a column containing an array of numbers to match the count of error messages:

| | |
|---|---|
| **Transformation Name** | `New formula` |
| **Parameter: Formula type** | `Single row formula` |
| **Parameter: Formula** | `range(0,arraylength_Errors,1)` |
| **Parameter: New column name** | `'range_Errors'` |

You can then use the `ARRAYZIP` function to zip together the two arrays into a single one:

| Transformation Name | `New formula` |
|---|---|
| Parameter: Formula type | `Single row formula` |
| Parameter: Formula | `arrayzip([range_Errors,Errors])` |
| Parameter: New column name | `'zipped_Errors'` |

The `unnest` transform uses the values in an array column as key values to break out rows in your dataset:

| Transformation Name | `Unnest Objects into columns` |
|---|---|
| Parameter: Column | `zipped_Errors` |

You might rename the above as `individual_Errors`. To clean up your dataset, you can now delete the following columns:

- `arraylength_Errors`
- `range_Errors`
- `zipped_Errors`

**Results:**

| Timestamp | Errors | individual_Errors |
|---|---|---|
| 02/16/16 15:31 | ["Unable to connect","File not found","Proxy down","conn. timeout"] | [0, "Unable to connect"] |
| 02/16/16 15:31 | ["Unable to connect","File not found","Proxy down","conn. timeout"] | [1, "File not found"] |
| 02/16/16 15:31 | ["Unable to connect","File not found","Proxy down","conn. timeout"] | [2, "Proxy down"] |
| 02/16/16 15:31 | ["Unable to connect","File not found","Proxy down","conn. timeout"] | [3, "conn. timeout"] |
| 02/16/16 15:29 | ["Access forbidden","Invalid password"] | [0, "Access forbidden"] |
| 02/16/16 15:29 | ["Access forbidden","Invalid password"] | [1, "Invalid password"] |

### Example - unnest test scores

The following example includes a `range` example to define a new index array.

This example illustrates you to use the flatten and unnest transforms.

**Source:**

You have the following data on student test scores. Scores on individual scores are stored in the `Scores` array, and you need to be able to track each test on a uniquely identifiable row. This example has two goals:

1. One row for each student test
2. Unique identifier for each student-score combination

| LastName | FirstName | Scores |
|---|---|---|
| Adams | Allen | [81,87,83,79] |
| Burns | Bonnie | [98,94,92,85] |
| Cannon | Charles | [88,81,85,78] |

**Transformation:**

When the data is imported from CSV format, you must add a `header` transform and remove the quotes from the `Scores` column:

| Transformation Name | Rename column with row(s) |
|---|---|
| **Parameter: Option** | Use row(s) as column names |
| **Parameter: Type** | Use a single row to name columns |
| **Parameter: Row number** | 1 |

| Transformation Name | Replace text or pattern |
|---|---|
| **Parameter: Column** | colScores |
| **Parameter: Find** | '\"' |
| **Parameter: Replace with** | '' |
| **Parameter: Match all occurrences** | true |

**Validate test date:** To begin, you might want to check to see if you have the proper number of test scores for each student. You can use the following transform to calculate the difference between the expected number of elements in the `Scores` array (4) and the actual number:

| Transformation Name | New formula |
|---|---|
| **Parameter: Formula type** | Single row formula |
| **Parameter: Formula** | (4 - arraylen(Scores)) |
| **Parameter: New column name** | 'numMissingTests' |

When the transform is previewed, you can see in the sample dataset that all tests are included. You might or might not want to include this column in the final dataset, as you might identify missing tests when the recipe is run at scale.

**Unique row identifier:** The `Scores` array must be broken out into individual rows for each test. However, there is no unique identifier for the row to track individual tests. In theory, you could use the combination of `LastName-FirstName-Scores` values to do so, but if a student recorded the same score twice, your dataset has duplicate rows. In the following transform, you create a parallel array called `Tests`, which contains an index array for the number of values in the `Scores` column. Index values start at `0`:

| Transformation Name | New formula |
|---|---|
| **Parameter: Formula type** | Single row formula |
| **Parameter: Formula** | range(0,arraylen(Scores)) |
| **Parameter: New column name** | 'Tests' |

Also, we will want to create an identifier for the source row using the `sourcerownumber` function:

| Transformation Name | New formula |
|---|---|

| | |
|---|---|
| **Parameter: Formula type** | `Single row formula` |
| **Parameter: Formula** | `sourcerownumber()` |
| **Parameter: New column name** | `'orderIndex'` |

**One row for each student test:** Your data should look like the following:

| LastName | FirstName | Scores | Tests | orderIndex |
|---|---|---|---|---|
| Adams | Allen | [81,87,83,79] | [0,1,2,3] | 2 |
| Burns | Bonnie | [98,94,92,85] | [0,1,2,3] | 3 |
| Cannon | Charles | [88,81,85,78] | [0,1,2,3] | 4 |

Now, you want to bring together the `Tests` and `Scores` arrays into a single nested array using the `arrayzip` function:

| | |
|---|---|
| **Transformation Name** | `New formula` |
| **Parameter: Formula type** | `Single row formula` |
| **Parameter: Formula** | `arrayzip([Tests,Scores])` |

Your dataset has been changed:

| LastName | FirstName | Scores | Tests | orderIndex | column1 |
|---|---|---|---|---|---|
| Adams | Allen | [81,87,83,79] | [0,1,2,3] | 2 | [[0,81],[1,87],[2,83],[3,79]] |
| Adams | Bonnie | [98,94,92,85] | [0,1,2,3] | 3 | [[0,98],[1,94],[2,92],[3,85]] |
| Cannon | Charles | [88,81,85,78] | [0,1,2,3] | 4 | [[0,88],[1,81],[2,85],[3,78]] |

Use the following to unpack the nested array:

| | |
|---|---|
| **Transformation Name** | `Expand arrays to rows` |
| **Parameter: Column** | `column1` |

Each test-score combination is now broken out into a separate row. The nested Test-Score combinations must be broken out into separate columns using the following:

| | |
|---|---|
| **Transformation Name** | `Unnest Objects into columns` |
| **Parameter: Column** | `column1` |
| **Parameter: Paths to elements** | `'[0]','[1]'` |

After you delete `column1`, which is no longer needed you should rename the two generated columns:

| | |
|---|---|
| **Transformation Name** | `Rename columns` |
| **Parameter: Option** | `Manual rename` |
| **Parameter: Column** | `column_0` |
| **Parameter: New column** | `'TestNum'` |

| | |
|---|---|
| **name** | |

| | |
|---|---|
| **Transformation Name** | `Rename columns` |
| **Parameter: Option** | `Manual rename` |
| **Parameter: Column** | `column_1` |
| **Parameter: New column name** | `'TestScore'` |

**Unique row identifier:** You can do one more step to create unique test identifiers, which identify the specific test for each student. The following uses the original row identifier `OrderIndex` as an identifier for the student and the `TestNumber` value to create the `TestId` column value:

| | |
|---|---|
| **Transformation Name** | `New formula` |
| **Parameter: Formula type** | `Single row formula` |
| **Parameter: Formula** | `(orderIndex * 10) + TestNum` |
| **Parameter: New column name** | `'TestId'` |

The above are integer values. To make your identifiers look prettier, you might add the following:

| | |
|---|---|
| **Transformation Name** | `Merge columns` |
| **Parameter: Columns** | `'TestId00','TestId'` |

**Extending:** You might want to generate some summary statistical information on this dataset. For example, you might be interested in calculating each student's average test score. This step requires figuring out how to properly group the test values. In this case, you cannot group by the `LastName` value, and when executed at scale, there might be collisions between first names when this recipe is run at scale. So, you might need to create a kind of primary key using the following:

| | |
|---|---|
| **Transformation Name** | `Merge columns` |
| **Parameter: Columns** | `'LastName','FirstName'` |
| **Parameter: Separator** | `'-'` |
| **Parameter: New column name** | `'studentId'` |

You can now use this as a grouping parameter for your calculation:

| | |
|---|---|
| **Transformation Name** | `New formula` |
| **Parameter: Formula type** | `Single row formula` |
| **Parameter: Formula** | `average(TestScore)` |
| **Parameter: Group rows by** | `studentId` |
| **Parameter: New column name** | `'avg_TestScore'` |

**Results:**

After you delete unnecessary columns and move your columns around, the dataset should look like the following:

| TestId | LastName | FirstName | TestNum | TestScore | studentId | avg_TestScore |
|--------|----------|-----------|---------|-----------|-----------|---------------|
| TestId0021 | Adams | Allen | 0 | 81 | Adams-Allen | 82.5 |
| TestId0022 | Adams | Allen | 1 | 87 | Adams-Allen | 82.5 |
| TestId0023 | Adams | Allen | 2 | 83 | Adams-Allen | 82.5 |
| TestId0024 | Adams | Allen | 3 | 79 | Adams-Allen | 82.5 |
| TestId0031 | Adams | Bonnie | 0 | 98 | Adams-Bonnie | 92.25 |
| TestId0032 | Adams | Bonnie | 1 | 94 | Adams-Bonnie | 92.25 |
| TestId0033 | Adams | Bonnie | 2 | 92 | Adams-Bonnie | 92.25 |
| TestId0034 | Adams | Bonnie | 3 | 85 | Adams-Bonnie | 92.25 |
| TestId0041 | Cannon | Chris | 0 | 88 | Cannon-Chris | 83 |
| TestId0042 | Cannon | Chris | 1 | 81 | Cannon-Chris | 83 |
| TestId0043 | Cannon | Chris | 2 | 85 | Cannon-Chris | 83 |
| TestId0044 | Cannon | Chris | 3 | 78 | Cannon-Chris | 83 |

# HOST Function

Finds the host value from a valid URL. Input values must be of URL or String type and can be literals or column references.

In this implementation, a host value includes everything from the end of the protocol identifier (if present) to the end of the extension (e.g. `.com`).

- For more information, see *Structure of a URL*.

**Wrangle vs. SQL:** This function is part of Wrangle , a proprietary data transformation language. Wrangle is not SQL. For more information, see *Wrangle Language*.

## Basic Usage

**URL literal examples:**

```
host('<span class="nolink">http://www.example.com</span>')
```

**Output:** Returns the value `www.example.com`.

**Column reference example:**

```
host(myURLs)
```

**Output:** Returns the host values extracted from the `myURLs` column.

## Syntax and Arguments

```
host(column_url)
```

| Argument | Required? | Data Type | Description |
|----------|-----------|-----------|-------------|
| column_url | Y | string | Name of column or String or URL literal containing the host value to extract |

For more information on syntax standards, see *Language Documentation Syntax Notes*.

**column_url**

Name of the column or URL or String literal whose values are used to extract the host value.

- Missing input values generate missing results.
- Multiple columns and wildcards are not supported.

**Usage Notes:**

| Required? | Data Type | Example Value |
|-----------|-----------|---------------|
| Yes | String literal or column reference (URL) | `http://www.example.com` |

# Examples

### Example - Domain, Host, Subdomain, and Suffix functions

This examples illustrates how you can extract component parts of a URL using specialized functions for the URL data type.

**Functions**:

| Item | Description |
|------|-------------|
| DOMAIN Function | Finds the value for the domain from a valid URL. Input values must be of URL or String type. |
| SUBDOMAIN Function | Finds the value a subdomain value from a valid URL. Input values must be of URL or String type. |
| HOST Function | Finds the host value from a valid URL. Input values must be of URL or String type and can be literals or column references. |
| SUFFIX Function | Finds the suffix value after the domain from a valid URL. Input values must be of URL or String type. |
| URLPARAMS Function | Extracts the query parameters of a URL into an Object. The Object keys are the parameter's names, and its values are the parameter's values. Input values must be of URL or String type. |
| FILTEROBJEC T Function | Filters the keys and values from an Object data type column based on a specified key value. |

**Source:**

Your dataset includes the following values for URLs:

| URL |
|-----|
| www.example.com |
| example.com/support |
| http://www.example.com/products/ |
| http://1.2.3.4 |
| https://www.example.com/free-download |
| https://www.example.com/about-us/careers |
| www.app.example.com |
| www.some.app.example.com |
| some.app.example.com |
| some.example.com |
| example.com |
| http://www.example.com?q1=broken%20record |
| http://www.example.com?query=khakis&app=pants |
| http://www.example.com?q1=broken%20record&q2=broken%20tape&q3=broken%20wrist |

**Transformation:**

When the above data is imported into the application, the column is recognized as a URL. All values are registered as valid, even the numeric address.

To extract the domain and subdomain values:

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | DOMAIN(URL) |
| Parameter: New column name | 'domain_URL' |

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | SUBDOMAIN(URL) |
| Parameter: New column name | 'subdomain_URL' |

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | HOST(URL) |
| Parameter: New column name | 'host_URL' |

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | SUFFIX(URL) |
| Parameter: New column name | 'suffix_URL' |

You can use the Pattern in the following transformation to extract protocol identifiers, if present, into a new column:

| Transformation Name | Extract text or pattern |
|---|---|
| Parameter: Column to extract from | URL |
| Parameter: Option | Custom text or pattern |
| Parameter: Text to extract | `` `{start}%*://` `` |

To clean this up, you might want to rename the column to `protocol_URL`.

To extract the path values, you can use the following regular expression:

> **NOTE:** Regular expressions are considered a developer-level method for pattern matching. Please use them with caution. See *Text Matching*.

| Transformation Name | Extract text or pattern |
|---|---|
| Parameter: Column to extract from | URL |
| Parameter: Option | Custom text or pattern |
| Parameter: Text to extract | /[^*:\/\/]\/.*$/ |

The above transformation grabs a little too much of the URL. If you rename the column to `path_URL`, you can use the following regular expression to clean it up:

| Transformation Name | Extract text or pattern |
|---|---|
| Parameter: Column to extract from | URL |
| Parameter: Option | Custom text or pattern |
| Parameter: Text to extract | /[!^\/].*$/ |

Delete the `path_URL` column and rename the `path_URL1` column to the deleted one. Then:

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | URLPARAMS(URL) |
| Parameter: New column name | 'urlParams' |

If you wanted to just see the values for the `q1` parameter, you could add the following:

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | FILTEROBJECT(urlParams,'q1') |
| Parameter: New column name | 'urlParam_q1' |

**Results:**

For display purposes, the results table has been broken down into separate sets of columns.

Column set 1:

| URL | host_URL | path_URL |
|---|---|---|
| www.example.com | www.example.com | |
| example.com/support | example.com | /support |
| http://www.example.com/products/ | www.example.com | /products/ |
| http://1.2.3.4 | 1.2.3.4 | |
| https://www.example.com/free-download | www.example.com | /free-download |

| | | |
|---|---|---|
| https://www.example.com/about-us/careers | www.example.com | /about-us /careers |
| www.app.example.com | www.app.example.com | |
| www.some.app.example.com | www.some.app.example. com | |
| some.app.example.com | some.app.example.com | |
| some.example.com | some.example.com | |
| example.com | example.com | |
| http://www.example.com?q1=broken%20record | www.example.com | |
| http://www.example.com?query=khakis&app=pants | www.example.com | |
| http://www.example.com?q1=broken%20record&q2=broken%20tape&q3=broken%20wrist | www.example.com | |

Column set 2:

| URL | protocol_URL | subdomain_URL | domain_URL | suffix_URL |
|---|---|---|---|---|
| www.example.com | | www | example | com |
| example.com/support | | | example | com |
| http://www.example.com/products/ | http:// | www | example | com |
| http://1.2.3.4 | http:// | | | |
| https://www.example.com/free-download | https:// | www | example | com |
| https://www.example.com/about-us/careers | https:// | www | example | com |
| www.app.example.com | | www.app | example | com |
| www.some.app.example.com | | www.some.app | example | com |
| some.app.example.com | | some.app | example | com |
| some.example.com | | some | example | com |
| example.com | | | example | com |
| http://www.example.com?q1=broken%20record | http:// | www | example | com |
| http://www.example.com?query=khakis&app=pants | http:// | www | example | com |
| http://www.example.com?q1=broken%20record&q2=broken%20tape&q3=broken%20wrist | http:// | www | example | com |

Column set 3:

| URL | urlParams | urlParam_q1 |
|---|---|---|
| www.example.com | | |
| example.com/support | | |
| http://www.example.com/products/ | | |
| http://1.2.3.4 | | |
| https://www.example.com/free-download | | |
| https://www.example.com/about-us/careers | | |
| www.app.example.com | | |
| www.some.app.example.com | | |

| | | |
|---|---|---|
| some.app.example.com | | |
| some.example.com | | |
| example.com | | |
| http://www.example.com?q1=broken%20record | {"q1":"broken record"} | {"q1":"broken record"} |
| http://www.example.com?query=khakis&app=pants | {"query":"khakis","app":"pants"} | |
| http://www.example.com?q1=broken%20record&q2=broken%20tape&q3=broken%20wrist | {"q1":"broken record", "q2":"broken tape", "q3":"broken wrist"} | {"q1":"broken record"} |

# DOMAIN Function

**Contents:**

---

Finds the value for the domain from a valid URL. Input values must be of URL or String type.

In this implementation, a domain value is all data between 1) the protocol identifier (if present) and the sub-domain and 2) the trailing, top-level domain information (e.g. `.com`).

- For more information, see *Structure of a URL* .
- You can also extract subdomain values by function. See *SUBDOMAIN Function*.

> **NOTE:** When the `DOMAIN` function parses a multi-tiered top-level domain such as `co.uk`, the output is the first part of the domain value (e.g. `co`). To return other parts of the domain information, you can use the `HOST` function. See *HOST Function*.

**Wrangle vs. SQL:** This function is part of Wrangle , a proprietary data transformation language. Wrangle is not SQL. For more information, see *Wrangle Language*.

## Basic Usage

**URL literal examples:**

```
domain(&apos;<span class="nolink">http://www.example.com</span>&apos;)
```

**Output:** Returns the value `example`.

```
domain(&apos;<span class="nolink">http://www.exampl.e.com</span>&apos;)
```

**Output:** Returns the value `e`.

**Column reference example:**

```
domain(myURLs)
```

**Output:** Returns the domain values extracted from the `myURLs` column.

## Syntax and Arguments

```
domain(column_url)
```

| Argument | Required? | Data Type | Description |
|---|---|---|---|
| column_url | Y | string | Name of column or String or URL literal containing the domain value to extract |

For more information on syntax standards, see *Language Documentation Syntax Notes*.

**column_url**

Name of the column or URL or String literal whose values are used to extract the domain value.

- Missing input values generate missing results.
- Multiple columns and wildcards are not supported.

**Usage Notes:**

| Required? | Data Type | Example Value |
|---|---|---|
| Yes | String literal or column reference (URL) | `http://www.example.com` |

## Examples

> **Tip:** For additional examples, see *Common Tasks*.

### Example - Filter out internal users

Here is some example web visitor information, including the name of the individual and the referring URL. You would like to filter out the internal users, whose referrer values include `test-value`.

| Name | Referrer |
|---|---|
| Joe Guy | http://www.example.com |
| Ian Holmes | http://www.test-value.com/support |
| Nick Knight | http://www.test-value.com |
| Axel Adams | http://www.example.com |
| Teri Towns | http://www.example.com/test-value |

**Transformation:**

The referrrer values include `test-value` as a non-domain value and varying URLs from the test-value.com domain. So, you should use the `DOMAIN` function to parse only the domain versions of these values. The following evaluates the `Referrer` column values for the `test-value` domain and generates true/false answers in a new column accordingly:

| Transformation Name | `New formula` |
|---|---|
| Parameter: Formula type | `Single row formula` |
| Parameter: Formula | `if(domain(Referrer)=='test-value',true,false)` |
| Parameter: New column name | `'isInternal'` |

Now that these values are flagged, you can filter out the internal names:

| Transformation Name | Filter rows |
|---|---|
| Parameter: Condition | Custom formula |
| Parameter: Type of formula | Custom single |
| Parameter: Condition | (isInternal == 'true') |
| Parameter: Action | Delete matching rows |

**Results:**

| Name | Referrer | isInternal |
|---|---|---|
| Joe Guy | http://www.example.com | false |
| Axel Adams | http://www.example.com | false |
| Teri Towns | http://www.example.com/test-value | false |

### Example - Domain, Subdomain, Host, and Suffix functions

This examples illustrates how you can extract component parts of a URL using specialized functions for the URL data type.

**Functions**:

| Item | Description |
|---|---|
| DOMAIN Function | Finds the value for the domain from a valid URL. Input values must be of URL or String type. |
| SUBDOMAIN Function | Finds the value a subdomain value from a valid URL. Input values must be of URL or String type. |
| HOST Function | Finds the host value from a valid URL. Input values must be of URL or String type and can be literals or column references. |
| SUFFIX Function | Finds the suffix value after the domain from a valid URL. Input values must be of URL or String type. |
| URLPARAMS Function | Extracts the query parameters of a URL into an Object. The Object keys are the parameter's names, and its values are the parameter's values. Input values must be of URL or String type. |
| FILTEROBJECT Function | Filters the keys and values from an Object data type column based on a specified key value. |

**Source:**

Your dataset includes the following values for URLs:

| URL |
|---|
| www.example.com |
| example.com/support |
| http://www.example.com/products/ |
| http://1.2.3.4 |
| https://www.example.com/free-download |
| https://www.example.com/about-us/careers |

| www.app.example.com |
|---|
| www.some.app.example.com |
| some.app.example.com |
| some.example.com |
| example.com |
| http://www.example.com?q1=broken%20record |
| http://www.example.com?query=khakis&app=pants |
| http://www.example.com?q1=broken%20record&q2=broken%20tape&q3=broken%20wrist |

**Transformation:**

When the above data is imported into the application, the column is recognized as a URL. All values are registered as valid, even the numeric address.

To extract the domain and subdomain values:

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | DOMAIN(URL) |
| Parameter: New column name | 'domain_URL' |

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | SUBDOMAIN(URL) |
| Parameter: New column name | 'subdomain_URL' |

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | HOST(URL) |
| Parameter: New column name | 'host_URL' |

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | SUFFIX(URL) |
| Parameter: New column name | 'suffix_URL' |

You can use the Pattern in the following transformation to extract protocol identifiers, if present, into a new column:

|  |  |
|---|---|

| Transformation Name | Extract text or pattern |
|---|---|
| Parameter: Column to extract from | URL |
| Parameter: Option | Custom text or pattern |
| Parameter: Text to extract | `` `{start}%*://` `` |

To clean this up, you might want to rename the column to `protocol_URL`.

To extract the path values, you can use the following regular expression:

> **NOTE:** Regular expressions are considered a developer-level method for pattern matching. Please use them with caution. See *Text Matching*.

| Transformation Name | Extract text or pattern |
|---|---|
| Parameter: Column to extract from | URL |
| Parameter: Option | Custom text or pattern |
| Parameter: Text to extract | `/[^*:\/\/]\/.*$/` |

The above transformation grabs a little too much of the URL. If you rename the column to `path_URL`, you can use the following regular expression to clean it up:

| Transformation Name | Extract text or pattern |
|---|---|
| Parameter: Column to extract from | URL |
| Parameter: Option | Custom text or pattern |
| Parameter: Text to extract | `/[!^\/].*$/` |

Delete the `path_URL` column and rename the `path_URL1` column to the deleted one. Then:

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | URLPARAMS(URL) |
| Parameter: New column name | 'urlParams' |

If you wanted to just see the values for the `q1` parameter, you could add the following:

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | FILTEROBJECT(urlParams,'q1') |
| Parameter: New column name | 'urlParam_q1' |

**Results:**

For display purposes, the results table has been broken down into separate sets of columns.

Column set 1:

| URL | host_URL | path_URL |
|---|---|---|
| www.example.com | www.example.com | |
| example.com/support | example.com | /support |
| http://www.example.com/products/ | www.example.com | /products/ |
| http://1.2.3.4 | 1.2.3.4 | |
| https://www.example.com/free-download | www.example.com | /free-download |
| https://www.example.com/about-us/careers | www.example.com | /about-us /careers |
| www.app.example.com | www.app.example.com | |
| www.some.app.example.com | www.some.app.example. com | |
| some.app.example.com | some.app.example.com | |
| some.example.com | some.example.com | |
| example.com | example.com | |
| http://www.example.com?q1=broken%20record | www.example.com | |
| http://www.example.com?query=khakis&app=pants | www.example.com | |
| http://www.example.com?q1=broken%20record&q2=broken%20tape&q3=broken% 20wrist | www.example.com | |

Column set 2:

| URL | protocol_URL | subdomain_URL | domain_URL | suffix_URL |
|---|---|---|---|---|
| www.example.com | | www | example | com |
| example.com/support | | | example | com |
| http://www.example.com/products/ | http:// | www | example | com |
| http://1.2.3.4 | http:// | | | |
| https://www.example.com/free-download | https:// | www | example | com |
| https://www.example.com/about-us/careers | https:// | www | example | com |
| www.app.example.com | | www.app | example | com |
| www.some.app.example.com | | www.some.app | example | com |
| some.app.example.com | | some.app | example | com |
| some.example.com | | some | example | com |
| example.com | | | example | com |
| http://www.example.com?q1=broken%20record | http:// | www | example | com |
| http://www.example.com?query=khakis&app=pants | http:// | www | example | com |
| http://www.example.com?q1=broken%20record&q2=broken% 20tape&q3=broken%20wrist | http:// | www | example | com |

Column set 3:

| URL | urlParams | urlParam_q1 |
|---|---|---|
| www.example.com | | |
| example.com/support | | |
| http://www.example.com/products/ | | |
| http://1.2.3.4 | | |
| https://www.example.com/free-download | | |
| https://www.example.com/about-us/careers | | |
| www.app.example.com | | |
| www.some.app.example.com | | |
| some.app.example.com | | |
| some.example.com | | |
| example.com | | |
| http://www.example.com?q1=broken%20record | {"q1":"broken record"} | {"q1":"broken record"} |
| http://www.example.com?query=khakis&app=pants | {"query":"khakis","app":"pants"} | |
| http://www.example.com?q1=broken%20record&q2=broken%20tape&q3=broken%20wrist | {"q1":"broken record", "q2":"broken tape", "q3":"broken wrist"} | {"q1":"broken record"} |

# SUBDOMAIN Function

Finds the value a subdomain value from a valid URL. Input values must be of URL or String type.

In this implementation, a subdomain value is the first group of characters in the URL after any protocol identifiers ( `http://` or `https://`, for example).

- For more information, see *Structure of a URL*.
- You can also extract domain values by function. See *DOMAIN Function*.

**Wrangle vs. SQL:** This function is part of Wrangle , a proprietary data transformation language. Wrangle is not SQL. For more information, see *Wrangle Language*.

## Basic Usage

**URL literal example:**

```
subdomain(&apos;<span class="nolink">http://www.example.com</span>&apos;)
```

**Output:** Returns the value: `www`.

```
subdomain(&apos;<span class="nolink">http://www.exampl.e.com</span>&apos;)
```

**Output:** Returns the value: `www.exampl`.

**Column reference example:**

```
subdomain(myURLs)
```

**Output:** Returns the subdomain values extracted from the `myURLs` column.

## Syntax and Arguments

```
subdomain(column_url)
```

| Argument | Required? | Data Type | Description |
|----------|-----------|-----------|-------------|
| column_url | Y | string | Name of column or String or URL literal containing the sub-domain value to extract |

For more information on syntax standards, see *Language Documentation Syntax Notes*.

**column_url**

Name of the column or URL or String literal whose values are used to extract the subdomain value.

- Missing input values generate missing results.
- Multiple columns and wildcards are not supported.

**Usage Notes:**

| Required? | Data Type | Example Value |
|-----------|-----------|---------------|
| Yes | String literal or column reference (URL) | |

```
                                        http://www.example.com
```

## Examples

> **Tip:** For additional examples, see *Common Tasks*.

### Example - Domain, Subdomain, Host, and Suffix functions

This examples illustrates how you can extract component parts of a URL using specialized functions for the URL data type.

**Functions**:

| Item | Description |
|------|-------------|
| DOMAIN Function | Finds the value for the domain from a valid URL. Input values must be of URL or String type. |
| SUBDOMAIN Function | Finds the value a subdomain value from a valid URL. Input values must be of URL or String type. |
| HOST Function | Finds the host value from a valid URL. Input values must be of URL or String type and can be literals or column references. |
| SUFFIX Function | Finds the suffix value after the domain from a valid URL. Input values must be of URL or String type. |
| URLPARAMS Function | Extracts the query parameters of a URL into an Object. The Object keys are the parameter's names, and its values are the parameter's values. Input values must be of URL or String type. |
| FILTEROBJECT Function | Filters the keys and values from an Object data type column based on a specified key value. |

**Source:**

Your dataset includes the following values for URLs:

| URL |
|-----|
| www.example.com |
| example.com/support |
| http://www.example.com/products/ |
| http://1.2.3.4 |
| https://www.example.com/free-download |
| https://www.example.com/about-us/careers |
| www.app.example.com |
| www.some.app.example.com |
| some.app.example.com |
| some.example.com |
| example.com |
| http://www.example.com?q1=broken%20record |
| http://www.example.com?query=khakis&app=pants |
| http://www.example.com?q1=broken%20record&q2=broken%20tape&q3=broken%20wrist |

**Transformation:**

When the above data is imported into the application, the column is recognized as a URL. All values are registered as valid, even the numeric address.

To extract the domain and subdomain values:

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | DOMAIN(URL) |
| Parameter: New column name | 'domain_URL' |

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | SUBDOMAIN(URL) |
| Parameter: New column name | 'subdomain_URL' |

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | HOST(URL) |
| Parameter: New column name | 'host_URL' |

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | SUFFIX(URL) |
| Parameter: New column name | 'suffix_URL' |

You can use the Pattern in the following transformation to extract protocol identifiers, if present, into a new column:

| Transformation Name | Extract text or pattern |
|---|---|
| Parameter: Column to extract from | URL |
| Parameter: Option | Custom text or pattern |
| Parameter: Text to extract | `` `{start}%*://` `` |

To clean this up, you might want to rename the column to `protocol_URL`.

To extract the path values, you can use the following regular expression:

> **NOTE:** Regular expressions are considered a developer-level method for pattern matching. Please use them with caution. See *Text Matching*.

| Transformation Name | Extract text or pattern |
|---|---|
| Parameter: Column to extract from | URL |
| Parameter: Option | Custom text or pattern |
| Parameter: Text to extract | /[^*:\/\/]\/.*$/ |

The above transformation grabs a little too much of the URL. If you rename the column to path_URL, you can use the following regular expression to clean it up:

| Transformation Name | Extract text or pattern |
|---|---|
| Parameter: Column to extract from | URL |
| Parameter: Option | Custom text or pattern |
| Parameter: Text to extract | /[!^\/].*$/ |

Delete the path_URL column and rename the path_URL1 column to the deleted one. Then:

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | URLPARAMS(URL) |
| Parameter: New column name | 'urlParams' |

If you wanted to just see the values for the q1 parameter, you could add the following:

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | FILTEROBJECT(urlParams,'q1') |
| Parameter: New column name | 'urlParam_q1' |

**Results:**

For display purposes, the results table has been broken down into separate sets of columns.

Column set 1:

| URL | host_URL | path_URL |
|---|---|---|
| www.example.com | www.example.com | |
| example.com/support | example.com | /support |
| http://www.example.com/products/ | www.example.com | /products/ |

| | | |
|---|---|---|
| http://1.2.3.4 | 1.2.3.4 | |
| https://www.example.com/free-download | www.example.com | /free-download |
| https://www.example.com/about-us/careers | www.example.com | /about-us /careers |
| www.app.example.com | www.app.example.com | |
| www.some.app.example.com | www.some.app.example. com | |
| some.app.example.com | some.app.example.com | |
| some.example.com | some.example.com | |
| example.com | example.com | |
| http://www.example.com?q1=broken%20record | www.example.com | |
| http://www.example.com?query=khakis&app=pants | www.example.com | |
| http://www.example.com?q1=broken%20record&q2=broken%20tape&q3=broken% 20wrist | www.example.com | |

Column set 2:

| URL | protocol_URL | subdomain_URL | domain_URL | suffix_URL |
|---|---|---|---|---|
| www.example.com | | www | example | com |
| example.com/support | | | example | com |
| http://www.example.com/products/ | http:// | www | example | com |
| http://1.2.3.4 | http:// | | | |
| https://www.example.com/free-download | https:// | www | example | com |
| https://www.example.com/about-us/careers | https:// | www | example | com |
| www.app.example.com | | www.app | example | com |
| www.some.app.example.com | | www.some.app | example | com |
| some.app.example.com | | some.app | example | com |
| some.example.com | | some | example | com |
| example.com | | | example | com |
| http://www.example.com?q1=broken%20record | http:// | www | example | com |
| http://www.example.com?query=khakis&app=pants | http:// | www | example | com |
| http://www.example.com?q1=broken%20record&q2=broken% 20tape&q3=broken%20wrist | http:// | www | example | com |

Column set 3:

| URL | urlParams | urlParam_q1 |
|---|---|---|
| www.example.com | | |
| example.com/support | | |
| http://www.example.com/products/ | | |
| http://1.2.3.4 | | |
| https://www.example.com/free-download | | |
| https://www.example.com/about-us/careers | | |
| | | |

| | | |
|---|---|---|
| www.app.example.com | | |
| www.some.app.example.com | | |
| some.app.example.com | | |
| some.example.com | | |
| example.com | | |
| http://www.example.com?q1=broken%20record | {"q1":"broken record"} | {"q1":"broken record"} |
| http://www.example.com?query=khakis&app=pants | {"query":"khakis","app":"pants"} | |
| http://www.example.com?q1=broken%20record&q2=broken%20tape&q3=broken%20wrist | {"q1":"broken record", "q2":"broken tape", "q3":"broken wrist"} | {"q1":"broken record"} |

# SUFFIX Function

Finds the suffix value after the domain from a valid URL. Input values must be of URL or String type.

This function is part of a set of functions for processing URL data.

- See *DOMAIN Function.*
- See *SUBDOMAIN Function.*
- For more information, see *Structure of a URL.*

**Wrangle vs. SQL:** This function is part of Wrangle , a proprietary data transformation language. Wrangle is not SQL. For more information, see *Wrangle Language.*

## Basic Usage

**URL literal examples:**

```
suffix(&apos;<span class="nolink">http://www.example.com</span>&apos;)
```

**Output:** Returns the value `com`.

```
suffix(&apos;<span class="nolink">http://www.exampl.e.com</span>&apos;)
```

**Output:** Returns the value `com`.

**Column reference example:**

```
suffix(myURLs)
```

**Output:** Returns the suffix values extracted from the `myURLs` column.

## Syntax and Arguments

```
suffix(column_url)
```

| Argument | Required? | Data Type | Description |
|----------|-----------|-----------|-------------|
| column_url | Y | string | Name of column or String or URL literal containing the suffix value to extract |

For more information on syntax standards, see *Language Documentation Syntax Notes.*

**column_url**

Name of the column or URL or String literal whose values are used to extract the suffix value.

- Missing input values generate missing results.
- Multiple columns and wildcards are not supported.

**Usage Notes:**

| Required? | Data Type | Example Value |
|-----------|-----------|---------------|

| Yes | String literal or column reference (URL) | `http://www.example.com` |

## Examples

> **Tip:** For additional examples, see *Common Tasks*.

**Example - Domain, Subdomain, Host, and Suffix functions**

This examples illustrates how you can extract component parts of a URL using specialized functions for the URL data type.

**Functions**:

| Item | Description |
|---|---|
| DOMAIN Function | Finds the value for the domain from a valid URL. Input values must be of URL or String type. |
| SUBDOMAIN Function | Finds the value a subdomain value from a valid URL. Input values must be of URL or String type. |
| HOST Function | Finds the host value from a valid URL. Input values must be of URL or String type and can be literals or column references. |
| SUFFIX Function | Finds the suffix value after the domain from a valid URL. Input values must be of URL or String type. |
| URLPARAMS Function | Extracts the query parameters of a URL into an Object. The Object keys are the parameter's names, and its values are the parameter's values. Input values must be of URL or String type. |
| FILTEROBJECT Function | Filters the keys and values from an Object data type column based on a specified key value. |

**Source:**

Your dataset includes the following values for URLs:

| URL |
|---|
| www.example.com |
| example.com/support |
| http://www.example.com/products/ |
| http://1.2.3.4 |
| https://www.example.com/free-download |
| https://www.example.com/about-us/careers |
| www.app.example.com |
| www.some.app.example.com |
| some.app.example.com |
| some.example.com |
| example.com |
| http://www.example.com?q1=broken%20record |
| http://www.example.com?query=khakis&app=pants |
| http://www.example.com?q1=broken%20record&q2=broken%20tape&q3=broken%20wrist |

**Transformation:**

When the above data is imported into the application, the column is recognized as a URL. All values are registered as valid, even the numeric address.

To extract the domain and subdomain values:

| Transformation Name | New formula |
| --- | --- |
| Parameter: Formula type | Single row formula |
| Parameter: Formula | DOMAIN(URL) |
| Parameter: New column name | 'domain_URL' |

| Transformation Name | New formula |
| --- | --- |
| Parameter: Formula type | Single row formula |
| Parameter: Formula | SUBDOMAIN(URL) |
| Parameter: New column name | 'subdomain_URL' |

| Transformation Name | New formula |
| --- | --- |
| Parameter: Formula type | Single row formula |
| Parameter: Formula | HOST(URL) |
| Parameter: New column name | 'host_URL' |

| Transformation Name | New formula |
| --- | --- |
| Parameter: Formula type | Single row formula |
| Parameter: Formula | SUFFIX(URL) |
| Parameter: New column name | 'suffix_URL' |

You can use the Pattern in the following transformation to extract protocol identifiers, if present, into a new column:

| Transformation Name | Extract text or pattern |
| --- | --- |
| Parameter: Column to extract from | URL |
| Parameter: Option | Custom text or pattern |
| Parameter: Text to extract | `` `{start}%*://` `` |

To clean this up, you might want to rename the column to `protocol_URL`.

To extract the path values, you can use the following regular expression:

| Transformation Name | Extract text or pattern |
|---|---|
| Parameter: Column to extract from | URL |
| Parameter: Option | Custom text or pattern |
| Parameter: Text to extract | /[^*:\/\/]\/.*$/ |

The above transformation grabs a little too much of the URL. If you rename the column to path_URL, you can use the following regular expression to clean it up:

| Transformation Name | Extract text or pattern |
|---|---|
| Parameter: Column to extract from | URL |
| Parameter: Option | Custom text or pattern |
| Parameter: Text to extract | /[!^\/].*$/ |

Delete the path_URL column and rename the path_URL1 column to the deleted one. Then:

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | URLPARAMS(URL) |
| Parameter: New column name | 'urlParams' |

If you wanted to just see the values for the q1 parameter, you could add the following:

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | FILTEROBJECT(urlParams,'q1') |
| Parameter: New column name | 'urlParam_q1' |

**Results:**

For display purposes, the results table has been broken down into separate sets of columns.

Column set 1:

| URL | host_URL | path_URL |
|---|---|---|
| www.example.com | www.example.com | |
| example.com/support | example.com | /support |
| http://www.example.com/products/ | www.example.com | /products/ |

| | | |
|---|---|---|
| http://1.2.3.4 | 1.2.3.4 | |
| https://www.example.com/free-download | www.example.com | /free-download |
| https://www.example.com/about-us/careers | www.example.com | /about-us /careers |
| www.app.example.com | www.app.example.com | |
| www.some.app.example.com | www.some.app.example. com | |
| some.app.example.com | some.app.example.com | |
| some.example.com | some.example.com | |
| example.com | example.com | |
| http://www.example.com?q1=broken%20record | www.example.com | |
| http://www.example.com?query=khakis&app=pants | www.example.com | |
| http://www.example.com?q1=broken%20record&q2=broken%20tape&q3=broken%20wrist | www.example.com | |

Column set 2:

| URL | protocol_URL | subdomain_URL | domain_URL | suffix_URL |
|---|---|---|---|---|
| www.example.com | | www | example | com |
| example.com/support | | | example | com |
| http://www.example.com/products/ | http:// | www | example | com |
| http://1.2.3.4 | http:// | | | |
| https://www.example.com/free-download | https:// | www | example | com |
| https://www.example.com/about-us/careers | https:// | www | example | com |
| www.app.example.com | | www.app | example | com |
| www.some.app.example.com | | www.some.app | example | com |
| some.app.example.com | | some.app | example | com |
| some.example.com | | some | example | com |
| example.com | | | example | com |
| http://www.example.com?q1=broken%20record | http:// | www | example | com |
| http://www.example.com?query=khakis&app=pants | http:// | www | example | com |
| http://www.example.com?q1=broken%20record&q2=broken%20tape&q3=broken%20wrist | http:// | www | example | com |

Column set 3:

| URL | urlParams | urlParam_q1 |
|---|---|---|
| www.example.com | | |
| example.com/support | | |
| http://www.example.com/products/ | | |
| http://1.2.3.4 | | |
| https://www.example.com/free-download | | |
| https://www.example.com/about-us/careers | | |
| | | |

| | | |
|---|---|---|
| www.app.example.com | | |
| www.some.app.example.com | | |
| some.app.example.com | | |
| some.example.com | | |
| example.com | | |
| http://www.example.com?q1=broken%20record | {"q1":"broken record"} | {"q1":"broken record"} |
| http://www.example.com?query=khakis&app=pants | {"query":"khakis","app":"pants"} | |
| http://www.example.com?q1=broken%20record&q2=broken%20tape&q3=broken%20wrist | {"q1":"broken record", "q2":"broken tape", "q3":"broken wrist"} | {"q1":"broken record"} |

# URLPARAMS Function

Extracts the query parameters of a URL into an Object. The Object keys are the parameter's names, and its values are the parameter's values. Input values must be of URL or String type.

This function is part of a set of functions for processing URL data.

- See *DOMAIN Function*.
- See *SUBDOMAIN Function*.
- See *SUFFIX Function*.
- For more information, see *Structure of a URL*.

**Wrangle vs. SQL:** This function is part of Wrangle , a proprietary data transformation language. Wrangle is not SQL. For more information, see *Wrangle Language*.

## Basic Usage

**URL literal examples:**

```
urlparams(&apos;<span class="nolink">http://example.com?color=blue&amp;shape=square<
/span>&apos;)
```

**Output:** Returns the following Object:

```
{"color": "blue","shape": "square"}
```

**Column reference example:**

```
urlparams(myURLs)
```

**Output:** Returns the query parameters extracted from the URLs in the `myURLs` column.

## Syntax and Arguments

```
urlparams(column_url)
```

| Argument | Required? | Data Type | Description |
|---|---|---|---|
| column_url | Y | string | Name of column or String or URL literal containing the parameters and their values to extract |

For more information on syntax standards, see *Language Documentation Syntax Notes*.

**column_url**

Name of the column or URL or String literal containing the query parameters and their values to extract.

- Missing input values generate missing results.
- Multiple columns and wildcards are not supported.

**Usage Notes:**

| Required? | Data Type | Example Value |
|---|---|---|
| Yes | String literal or column reference (URL) | `http://www.example.com?q1=hello&q2=goodbye` |

## Examples

> **Tip:** For additional examples, see *Common Tasks*.

### Example - Domain, Subdomain, Host, and Suffix functions

This examples illustrates how you can extract component parts of a URL using specialized functions for the URL data type.

**Functions**:

| Item | Description |
|---|---|
| DOMAIN Function | Finds the value for the domain from a valid URL. Input values must be of URL or String type. |
| SUBDOMAIN Function | Finds the value a subdomain value from a valid URL. Input values must be of URL or String type. |
| HOST Function | Finds the host value from a valid URL. Input values must be of URL or String type and can be literals or column references. |
| SUFFIX Function | Finds the suffix value after the domain from a valid URL. Input values must be of URL or String type. |
| URLPARAMS Function | Extracts the query parameters of a URL into an Object. The Object keys are the parameter's names, and its values are the parameter's values. Input values must be of URL or String type. |
| FILTEROBJECT Function | Filters the keys and values from an Object data type column based on a specified key value. |

**Source:**

Your dataset includes the following values for URLs:

| URL |
|---|
| www.example.com |
| example.com/support |
| http://www.example.com/products/ |
| http://1.2.3.4 |
| https://www.example.com/free-download |
| https://www.example.com/about-us/careers |
| www.app.example.com |
| www.some.app.example.com |
| some.app.example.com |
| some.example.com |

| | |
|---|---|
| example.com | |

| | |
|---|---|
| http://www.example.com?q1=broken%20record | |

| | |
|---|---|
| http://www.example.com?query=khakis&app=pants | |

| | |
|---|---|
| http://www.example.com?q1=broken%20record&q2=broken%20tape&q3=broken%20wrist | |

**Transformation:**

When the above data is imported into the application, the column is recognized as a URL. All values are registered as valid, even the numeric address.

To extract the domain and subdomain values:

| Transformation Name | New formula |
|---|---|
| **Parameter: Formula type** | Single row formula |
| **Parameter: Formula** | DOMAIN(URL) |
| **Parameter: New column name** | 'domain_URL' |

| Transformation Name | New formula |
|---|---|
| **Parameter: Formula type** | Single row formula |
| **Parameter: Formula** | SUBDOMAIN(URL) |
| **Parameter: New column name** | 'subdomain_URL' |

| Transformation Name | New formula |
|---|---|
| **Parameter: Formula type** | Single row formula |
| **Parameter: Formula** | HOST(URL) |
| **Parameter: New column name** | 'host_URL' |

| Transformation Name | New formula |
|---|---|
| **Parameter: Formula type** | Single row formula |
| **Parameter: Formula** | SUFFIX(URL) |
| **Parameter: New column name** | 'suffix_URL' |

You can use the Pattern in the following transformation to extract protocol identifiers, if present, into a new column:

| Transformation Name | Extract text or pattern |
|---|---|
| **Parameter: Column to extract from** | URL |
| **Parameter: Option** | Custom text or pattern |
| | `` `{start}%*://` `` |

| | |
|---|---|
| **Parameter: Text to extract** | |

To clean this up, you might want to rename the column to `protocol_URL`.

To extract the path values, you can use the following regular expression:

> **NOTE:** Regular expressions are considered a developer-level method for pattern matching. Please use them with caution. See *Text Matching*.

| | |
|---|---|
| **Transformation Name** | `Extract text or pattern` |
| **Parameter: Column to extract from** | `URL` |
| **Parameter: Option** | `Custom text or pattern` |
| **Parameter: Text to extract** | `/[^*:\/\/]\/.*$/` |

The above transformation grabs a little too much of the URL. If you rename the column to `path_URL`, you can use the following regular expression to clean it up:

| | |
|---|---|
| **Transformation Name** | `Extract text or pattern` |
| **Parameter: Column to extract from** | `URL` |
| **Parameter: Option** | `Custom text or pattern` |
| **Parameter: Text to extract** | `/[!^\/].*$/` |

Delete the `path_URL` column and rename the `path_URL1` column to the deleted one. Then:

| | |
|---|---|
| **Transformation Name** | `New formula` |
| **Parameter: Formula type** | `Single row formula` |
| **Parameter: Formula** | `URLPARAMS(URL)` |
| **Parameter: New column name** | `'urlParams'` |

If you wanted to just see the values for the `q1` parameter, you could add the following:

| | |
|---|---|
| **Transformation Name** | `New formula` |
| **Parameter: Formula type** | `Single row formula` |
| **Parameter: Formula** | `FILTEROBJECT(urlParams,'q1')` |
| **Parameter: New column name** | `'urlParam_q1'` |

**Results:**

For display purposes, the results table has been broken down into separate sets of columns.

Column set 1:

| URL | host_URL | path_URL |
|---|---|---|
| www.example.com | www.example.com | |
| example.com/support | example.com | /support |
| http://www.example.com/products/ | www.example.com | /products/ |
| http://1.2.3.4 | 1.2.3.4 | |
| https://www.example.com/free-download | www.example.com | /free-download |
| https://www.example.com/about-us/careers | www.example.com | /about-us /careers |
| www.app.example.com | www.app.example.com | |
| www.some.app.example.com | www.some.app.example. com | |
| some.app.example.com | some.app.example.com | |
| some.example.com | some.example.com | |
| example.com | example.com | |
| http://www.example.com?q1=broken%20record | www.example.com | |
| http://www.example.com?query=khakis&app=pants | www.example.com | |
| http://www.example.com?q1=broken%20record&q2=broken%20tape&q3=broken%20wrist | www.example.com | |

Column set 2:

| URL | protocol_URL | subdomain_URL | domain_URL | suffix_URL |
|---|---|---|---|---|
| www.example.com | | www | example | com |
| example.com/support | | | example | com |
| http://www.example.com/products/ | http:// | www | example | com |
| http://1.2.3.4 | http:// | | | |
| https://www.example.com/free-download | https:// | www | example | com |
| https://www.example.com/about-us/careers | https:// | www | example | com |
| www.app.example.com | | www.app | example | com |
| www.some.app.example.com | | www.some.app | example | com |
| some.app.example.com | | some.app | example | com |
| some.example.com | | some | example | com |
| example.com | | | example | com |
| http://www.example.com?q1=broken%20record | http:// | www | example | com |
| http://www.example.com?query=khakis&app=pants | http:// | www | example | com |
| http://www.example.com?q1=broken%20record&q2=broken%20tape&q3=broken%20wrist | http:// | www | example | com |

Column set 3:

| URL | urlParams | urlParam_q1 |
|---|---|---|
| www.example.com | | |
| example.com/support | | |

| | | |
|---|---|---|
| http://www.example.com/products/ | | |
| http://1.2.3.4 | | |
| https://www.example.com/free-download | | |
| https://www.example.com/about-us/careers | | |
| www.app.example.com | | |
| www.some.app.example.com | | |
| some.app.example.com | | |
| some.example.com | | |
| example.com | | |
| http://www.example.com?q1=broken%20record | {"q1":"broken record"} | {"q1":"broken record"} |
| http://www.example.com?query=khakis&app=pants | {"query":"khakis","app":"pants"} | |
| http://www.example.com?q1=broken%20record&q2=broken%20tape&q3=broken%20wrist | {"q1":"broken record", "q2":"broken tape", "q3":"broken wrist"} | {"q1":"broken record"} |

# Other Language Topics

# Text Matching

**Contents:**

This section enables you to work with text matching in Designer Cloud Powered by Trifacta® Enterprise Edition application.

## Match Types

Designer Cloud Powered by Trifacta® Enterprise Edition supports the following types of text matching clauses:

- **String literals** match specified strings exactly. Written using single quotes ('...') or double quotes ("...").
- **Regular expressions** enable pattern-based matching. Regular expressions are written using forward slashes (/.../). The syntax is based on *RE2* and *PCRE* regular expressions.

> **NOTE:** Regular expressions are considered a developer-level capability and can have significant consequences if they are improperly specified. Unless you are comfortable with regular expressions, you should use Patterns instead.

- **Patterns**  are custom selectors for patterns in your data and provide a simpler and more readable alternative to regular expressions. They are written using backticks (`...`).

- **Column names** are simple text strings in  Wrangle . If the column name contains a space, it must be bracketed in curly braces: `{my Column Name}`.

## Trifacta Patterns Syntax

The following tables contain syntax information about  Patterns :

> **Tip:** You can use Patterns as parameters in your flow. You assign the pattern to a variable, which can be used in your recipe steps. For more information, see *Manage Parameters Dialog*.

> **Tip:** After using Patterns , regular expressions, or string literals in a recipe step, you can reuse them in your transformations where applicable.

### Character patterns

These patterns apply to single characters and strings of characters

| Pattern | Description |
|---|---|

| | |
|---|---|
| % | Match any character, exactly once |
| %? | Match any character, zero or one times |
| %* | Match any character, zero or more times |
| %+ | match any character, one or more times |
| %{3} | Match any character, exactly three times |
| %{3,5} | Match any character, 3, 4, or 5 times |
| # | Digit character [0-9] |
| {any} | Match any character, exactly once |
| {alpha} | Alpha character [A-Za-z_] |
| {upper} | Uppercase alpha character [A-Z_] |
| {lower} | Lowercase alpha character [a-z_] |
| {digit} | Digit character [0-9] |
| {delim} | Single delimiter character e.g :, ,, \|, /, -, ., \s |
| {delim-ws} | Single delimiter and all the whitespace around it |
| {alpha-numeric} | Match a single alphanumeric character |
| {alphanum-underscore} | Match a single alphanumeric character or underscore character |
| {at-username} | Match @username values |
| {hashtag} | Match #hashtag values |
| {hex} | Match hexadecimal number (e.g. 2FA3) |

## Position patterns

These patterns describe positions relative to the entire string.

| Pattern | Description |
|---|---|
| {start} | Match the start of the line |
| {end} | Match the end of the line |

## Type patterns

These patterns can be used to match strings that fit a particular data type, except for Datetime patterns.

| Pattern | Description |
|---|---|
| {phone} | Match a valid U.S. phone number. |
| {email} | Match a valid email address. |
| {url} | Match a valid URL. |
| {ip-address} | Match a valid IP address. |
| {hex-ip-address} | Match a valid hexadecimal IP address (e.g. 0x0CA40012) |
| {bool} | Match a valid Boolean value. |
| {street} | Match a U.S.-formatted street address (e.g. 123 Main Street) |
| {occupancy} | Match a valid U.S.-formatted occupancy address value (e.g. Apt 2D) |
| {city} | Match a city name within U.S.-formatted address value |

| | |
|---|---|
| {state} | Match a valid U.S. state value (e.g. California). |
| {state-abbrev} | Match a valid two-letter U.S. state abbreviation value (e.g. CA) |
| {zip} | Match a valid five-digit zip code |

## Datetime patterns

| Pattern | Description |
|---|---|
| {month} | Match full name of month (e.g. January) |
| {month-abbrev} | Match short name of month (e.g. Jan) |
| {time} | Match time value in HOUR:MINUTE:SECOND format (e.g. 11:59:23) |
| {period} | Match time period of the day: AM/PM |
| {dayofweek} | Match long name for day of the week (e.g. Sunday). |
| {dayofweek-abbrev} | Match short name for day of the week (e.g. Sun). |
| {utcoffset} | Match a valid UTC offset value (e.g. -0500, +0400, Z) |

> **NOTE:** You can use the Datetime data type formatting tokens as part of your Patterns to build a variety of matching patterns for date and time values.

## Grouping patterns

| Pattern | Description |
|---|---|
| {[...]} | character class matches characters in brackets |
| {![...]} | negated class matches characters not in brackets |
| (...) | grouping, including captures |
| #, %, ?, *, +, {, }, (, ), \, ', \n, \t | escaped characters or pattern modifiers Use a double backslash (\\) to denote an escaped string literal. For more information, see *Escaping Strings in Transformations*. |
| \| | logical OR |

- Logical AND is the implied operator when you concatenate text matching patterns.
- Logical NOT is managed using negated classes.

See also *Capture Group References*.

## Patterns  Examples

### Basic

Match first three characters:

```
`{start}%{3}`
```

Match last four letters (numeric or other character types do not match):

```
`{alpha}{4}{end}`
```

Match first word:

```
`{start}{alpha}+`
```

Matches date values in general YYYY*MM*dd format:

```
`{yyyy}{delim}{MM}{delim}{dd}`
```

Matches time values in 12-hour format:

```
`{h}{delim}{mm}{delim}{s}`
```

### In transformations

The following transformation masks credit card number patterns, except for the last four digits:

| | |
|---|---|
| **Transformation Name** | `Replace text or patterns` |
| **Parameter: Columns** | `myCreditCardNumbers` |
| **Parameter: Find** | `` `{start}{digit}{4}{any}{digit}{4}{any}{digit}{4}{any}({digit}{4}){end}` `` |
| **Parameter: Replace with** | `XXXX-XXXX-XXXX-$1` |

**Notes:**

- The inclusion of the `{start}` and `{end}` tokens assures that the matches are made only when the pattern is found across the entire value in a cell.
- The parenthesis in the Find value identify the capture group, which is referenced in the Replace With value as `$1`. See *Capture Group References.*

The above transformation matches values based on the structure of the data, instead of the data type.

- Some values that follow this pattern are not valid credit card numbers, so it's meaningful to check against the data type.
- If for some reason, you have values that are not credit card numbers yet follow the credit card pattern, those values will be masked as well by this transformation.

So to be safe, you might try the following set of transformations to ensure that you are matching on credit card values.

**Step 1:** If the number in your source column is valid, write it to a new column.

| | |
|---|---|
| **Transformation Name** | `New formula` |
| **Parameter: Formula type** | `Single row formula` |
| **Parameter: Formula** | `IFVALID(myCreditCardNumbers,'Creditcard'),$col,'')` |
| **Parameter: New column name** | `myCreditCardNumbersMasked` |

Notes:

- The IFVALID function tests to see if a set of values is valid for a specified data type, `'Creditcard'` in this case. For more information on the strings that you can use to test against data type, see *Valid Data Type Strings*.

- The `$col` is a reference to the value in the column where the evaluation is being performed. For more information, see *Source Metadata References*.

**Step 2:** The `myCreditCardNumbersMasked` column now contains values that are valid credit card numbers from your source column. You can now apply the masking step.

| Transformation Name | `Replace text or patterns` |
|---|---|
| **Parameter: Columns** | `myCreditCardNumbersMasked` |
| **Parameter: Find** | `` `{start}{digit}{4}{any}{digit}{4}{any}{digit}{4}{any}({digit}{4}){end}` `` |
| **Parameter: Replace with** | `XXXX-XXXX-XXXX-$1` |

**Step 3:** If needed, you can move the masked values back to the source column.

| Transformation Name | `Edit column with formula` |
|---|---|
| **Parameter: Columns** | `myCreditCardNumbers` |
| **Parameter: Formula** | `IF(myCreditCardNumbersMasked<>'', myCreditCardNumbersMasked,'')` |

The `myCreditCardNumbers` column now contains only valid credit card numbers that have been asked. The application is likely to infer the data type of the column as String.

Delete the `myCreditCardNumbersMasked` column.

# Escaping Strings in Transformations

This section describes how to escape strings in your transformations.

In the platform, the backslash character (\) is used to escape values within strings. The character following the escaping character is treated as a string literal.

For example, the following value is used to represent a matching value of & only:

```
`\&`
```

Escaping can be applied to parameters in functions. For example, in the data grid, you have the following values in a column:

| MyStringCol |
| --- |
| This works. |
| You can't break this. |
| Not broken yet. |

To find the value `can't`, you could enter the following pattern:

| Transformation Name | New formula |
| --- | --- |
| Parameter: Formula type | Single row formula |
| Parameter: Formula | FIND(MyStringCol, 'can\'t',true,0) |
| Parameter: New column name | 'MyFindResults' |

The above transformation results in the following:

| MyStringCol | MyFindResults |
| --- | --- |
| This works. | |
| You can't break this. | 4 |
| Not broken yet. | |

All pattern type markers can be escaped if using the marking character in a string:

| Pattern type | Marker | Escaped character |
| --- | --- | --- |
| literal value | ' | \' |
| Pattern | ` | \` |
| Regular expression | / | \/ |

**A note on JSON:**

In the data grid, JSON Objects and arrays include additional escaping to show that the values are strings. For example, the data grid shows:

```
{"re\"becca","hello"}
```

The first JSON element displayed in the GUI is `re\"becca`, but the desired match is `re"becca`.

> **Tip:** For best results in pattern matching, you should make selections in the data grid and modify if necessary.

Below, you can see how this JSON pattern is specified in the following example transformation:

| Transformation Name | `Pivot columns` |
|---|---|
| **Parameter: Column** | `MyCol` |
| **Parameter: Paths to elements** | `[\"re\\\"becca\"]` |

- The `keys` value must be single-quoted. Since the keys are specified for Object data, the square bracket notation is used.
- Within the square brackets, the individual keys must be double-quoted.
- The first two backslashes (`\\`) indicate that you are escaping a single backslash character.
- The third backslash indicates that you are escaping the double-quote that is part of the string to match.

In the following example, you are trying to match on the above string, including the double-quotes around it: `"re\"becca"`.

| Transformation Name | `Pivot columns` |
|---|---|
| **Parameter: Column** | `MyCol` |
| **Parameter: Paths to elements** | `'[\"re\\\"becca\"]'` |

The bracketing double-quotes must be escaped, too.

# Pattern Clause Position Matching

**Contents:**

For a number of different transform types, you can specify the limits at which any match is valid for a text string. In the diagram below, you can see how six different positional identifiers can be applied to pattern matching:

> **NOTE:** Depending on the type of transform, some of these clauses are not available.

## Positioning



*Figure: Pattern Clause Positioning*

**after**

Identifies pattern or string after which the match is evaluated.

**Example transformation:**

| Transformation Name | Extract text or pattern |
|---|---|
| Parameter: Column | MySentence |
| Parameter: Option | On pattern |
| Parameter: After pattern | 'eat ' |

**Extracts:**

```
pizza on Fridays with my friends."
```

## from

Identifies pattern or string from which the match is evaluated. Any match includes the `from` clause pattern or string.

**Example transformation:**

| | |
|---|---|
| Transformation Name | `Extract text or pattern` |
| Parameter: Column | `MySentence` |
| Parameter: Option | `Between two parameters` |
| Parameter: After pattern | `'eat '` |
| Parameter: Include as part of match | `true` |

**Extracts:**

```
eat pizza on Fridays with my friends."
```

## before

Identifies pattern or string before which the match is evaluated.

**Example transformation:**

| | |
|---|---|
| Transformation Name | `Extract text or pattern` |
| Parameter: Column | `MySentence` |
| Parameter: Option | `On pattern` |
| Parameter: Before pattern | `'friends'` |

**Extracts:**

```
"I like to eat pizza on Fridays with my
```

## to

Identifies pattern or string up to which the match is evaluated. Any match includes the `to` clause pattern or string.

**Example transformation:**

| | |
|---|---|
| Transformation Name | `Extract text or pattern` |
| Parameter: Column | `MySentence` |
| Parameter: Option | `Between two patterns` |
| Parameter: Before pattern | `'friends'` |
| | |

| | |
|---|---|
| **Parameter: Include as part of match** | `true` |

**Extracts:**

```
"I like to eat pizza on Fridays with my friends
```

### on

Identifies pattern or string in which the match may be found.

**Example transformation:**

| Transformation Name | `Extract text or pattern` |
|---|---|
| **Parameter: Column** | `MySentence` |
| **Parameter: Option** | `On pattern` |
| **Parameter: Match pattern** | `'Fridays'` |

**Extracts:**

```
Fridays
```

### at

Identifies the index of starting (x) and ending (y) characters in the string to match. In the above example, `at:2,6` matches the string `like`.

**Example transformation:**

| Transformation Name | `Extract text or pattern` |
|---|---|
| **Parameter: Column** | `MySentence` |
| **Parameter: Option** | `Between two positions` |
| **Parameter: Positions** | `2,6` |

**Extracts:**

```
like
```

## Pattern Parameter Interactions

The following table identifies the pattern parameters that can be matched with the parameter in the left column.

> **NOTE:** The `at` parameter does not interact with any of the listed parameters.

| Parameter | after | from | before | to | on |
|-----------|-------|------|--------|-----|-----|
| `after` | See Note 1. | No. | `before` or `to` | `before` or `to` | Yes. Can include `before`. |
| `from` | No. | See Note 1. | `before` or `to` | `before` or `to` | No. |
| `before` | `after` or `from` | `after` or `from` | See Note 1. | No. | Yes. Can include `after`. |
| `to` | `after` or `from` | `after` or `from` | No. | See Note 1. | No. |
| `on` | Yes. Can include `before`. | No. | Yes. Can include `after`. | No. | |

- **Note 1:** If there is no other pattern parameter in the transform, the maximum number of matches per cell is 1. If there is a matching parameter, more matches per cell can be found.

# String Collation Rules

**Collation** refers to the organizing of written content into a standardized order. String comparison functions utilize collation rules for Latin. A summary of the rules:

- Comparisons are case-sensitive.
    - Uppercase letters are greater than lowercase versions of the same letter.
    - However, lowercase letters that are later in the alphabet are greater than the uppercase version of the previous letter.
- Two strings are equal if they match identically.
    - If two strings are identical except that the second string contains one additional character at the end, the second string is greater.
- A **normalized version** of a letter is the unaccented, lowercase version of the letter. In string comparison, it is the lowest value of all of its variants.
    - a is less than .
    - However, when compared to b, a = .
    - The set of Latin normalized characters contains more than 26 characters.

This table illustrates some generalized rules of Latin collation.

| Order | Description | Lesser Example | Greater Example |
|-------|-------------|----------------|-----------------|
| 1 | whitespace | (space) | (return) |
| 2 | Punctuation | ' | @ |
| 3 | Digits | 1 | 2 |
| 4 | Letters | a | A |
| 5 | | A | b |

**Resources:**

> **NOTE:** In the following set of charts (linked below), the values at the top of the page are lower than the values listed lower on the page. Similarly, the charts listed in the left nav bar are listed in ascending order.

For more information on the applicable collation rules, see *http://www.unicode.org/charts/collation/*.

# Supported Special Regular Expression Characters

**Contents:**

Designer Cloud Powered by Trifacta® Enterprise Edition supports a set of special characters for regular expressions that are common to all of the execution engines supported by the platform.

## Slashes

The forward slash character is used to denote the boundaries of the regular expression:

```
/this_is_my_regular_expression/
```

- The backslash character (\) is the escaping character. It can be used to denote an escaped character, a string, literal, or one of the set of supported special characters.
- Use a double backslash (\\) to denote an escaped string literal. For more information, see *Escaping Strings in Transformations*.

## Supported Special RegEx Characters

The table below identifies the special characters that are supported in the platform.

| Special Characters | Description |
|---|---|
| \\ | String literal match for \ character. |
| \b | Matches any zero-width word boundary, such as between a letter and a space. <br><br> Example: /\bre/ does not match re in tire , since re is not on the word boundary. /re\b/ does match. |
| \B | Matches any zero-width non-word boundary, such as between two letters or two spaces. <br><br> Example: /\Bre/ matches re in tire. It does not match in respect, since that instance of re is on a word boundary. |
| \cX | Matches a control character (CTRL + A-Z), where X is the corresponding letter in the alphabet. |
| \d | Matches any digit. |
| \D | Matches any non-digit. |
| \f | Matches a form feed. |
| \n | Matches a line feed. <br><br> **NOTE:** These characters are not supported in inputs for Object and Array data types. |
| \r | Matches a carriage return. |
| \s | Matches any whitespace character. These characters include: |

| | |
|---|---|
| | - space<br>- tab<br>- form feed<br>- line feed<br>- Other Unicode space characters |
| \S | Matches any character that is not one of the supported whitespace characters. |
| \t | Matches a horizontal tab.<br><br>> **NOTE:** These characters are not supported in inputs for Object and Array data types. |
| \v | Matches a vertical tab. |
| \w | Matches any alphanumeric value, including the underscore.<br><br>> **Tip:** Column names must match the same set of characters. |
| \W | Matches any non-alphanumeric character, including the underscore. |
| \xHH | Matches the ASCII character code as expressed by the hexadecimal value HH. |
| \uHHHH | Matches the Unicode character code as expressed by the hexadecimal value HHHH . |

## Required Escaped Characters

The following characters have special meaning within a regular expression.

```
. ^ $ * + - ? ( ) [ ] { } \ | - /
```

To reference the literal character, you must escape it within the regular expression, as in:

```
/\./
```

# Capture Group References

**Contents:**

---

In  Wrangle  transformations that support use of patterns, you may need to specify capture groups. A **capture group** is a pattern that describes a set of one or more characters that constitute a match. These matches can be programmatically referenced in replacement values.

- These patterns are described using regular expression syntax.  Designer Cloud Powered by Trifacta Enterprise Edition implements a version of regular expressions based off of *RE2* and *PCRE* regular expressions.

## Basic Capture Groups

### Example 1

| Transformation Name | `Replace text or pattern` |
|---|---|
| **Parameter: Columns** | `All` |
| **Parameter: Find** | `` `{start}(%+) ` `` |
| **Parameter: Replace with** | `'First Word\:$1'` |

**Elements of the matching pattern (on:):**

| Reference | Description |
|---|---|
| {start} | A Pattern reference to the start of the tested value. |
| (%+) | Matches on one or more characters of any time. <br><br> > **NOTE:** The parentheses indicate that this set of characters is a capture group. |
|  | Last character in the matching pattern is an empty space. |

**Matches:** First set of any characters in the tested value up to the first empty space (the first word), across all columns of the dataset.

**Replaced with:** The text value `First Word:` followed by a reference to the first capture group (`$1`), which returns the first word found in the tested value.

**Example 2**

The previous example works fine, as long as there is a space in the tested value to identify the end of the first word. If there is one and only word in the tested value, then you must amend the `on:` parameter value with the following:

| | |
|---|---|
| **Transformation Name** | `Replace text or pattern` |
| **Parameter: Columns** | `All` |
| **Parameter: Find** | `` `{start}(%+) ( |{end})` `` |
| **Parameter: Replace with** | `'First Word\:$1'` |

In this case, the second capture group features two elements:

| Reference | Description |
|---|---|
| | first character in the second capture group is an empty space. |
| \| | Logical OR, which means that the capture group matches on either the empty space or the following value, which is a reference to the end of the tested value. |
| {end} | A Pattern reference to the end of the tested value. |

**Example 3**

| | |
|---|---|
| **Transformation Name** | `Replace text or pattern` |
| **Parameter: Columns** | `All` |
| **Parameter: Find** | `` `{start}(%+) (%+)( |{end})` `` |
| **Parameter: Replace with** | `'Second Word\:$2'` |

**Matches:** The `on:` pattern has been augmented to include the second word in the tested value, across all columns of the dataset.

**Replaced with:** The text value `Second Word:` followed by a reference to the second capture group (`$2`), which returns the second word found in the tested value.

## Dollar sign in Replace transformation

The dollar sign ($) is used as a form of escape character in the `with` parameter of the Replace transformation. This pattern identifies the replacement string.

In the table below, you can review how these replacement patterns are supported.

| Pattern | Description |
|---|---|
| `$$` | Inserts a $ in the replacement value. |
| `$n` or `$nn` | For non-negative digits `n`, this pattern inserts the nth parameterized sub-match string, provided that the first argument was a regex object. |

**Examples**

In the following example, the `MyColumn` column contains the value `foobar` in all rows.

| source value | Replace transformation | | replacement |
|---|---|---|---|
| `foobar` | **Transformation Name** | `Replace text or pattern` | `$foobar` |
| | **Parameter: Column** | `MyColumn` | |
| | **Parameter: Find** | `'f'` | |
| | **Parameter: Replace with** | `'$$f'` | |
| `foobar` | **Transformation Name** | `Replace text or pattern` | `o` |
| | **Parameter: Column** | `MyColumn` | |
| | **Parameter: Find** | `` `(f)(o)o(b)ar` `` | |
| | **Parameter: Replace with** | `'$2'` | |
| | Note that the `on` parameter is a Pattern . | | |

## Positive and Negative Lookaheads

In regular expressions, you can use positive and negative lookahead capture groups to capture content that is conditionally followed or not followed by a specified capture group.

| Type | Example expression | |
|---|---|---|
| Positive lookahead | `/q(?u)/` | Capture the letter q only when it is followed by the letter u. Letter u is not captured. |
| Negative lookahead | `/q(?!u)/` | Capture the letter q when it is not followed by the letter u. Letter u is not captured. |

# Valid Data Type Strings

When referencing a data type within a transform, you can use the following strings to identify each type:

> **NOTE:** In Wrangle transforms, these values are case-sensitive.

> **NOTE:** When specifying a data type by name, you must use the String value listed below. The Data Type value is the display name for the type.

| Data Type | String |
|---|---|
| String | `'String'` |
| Integer | `'Integer'` |
| Decimal | `'Float'` |
| Boolean | `'Bool'` |
| Social Security Number | `'SSN'` |
| Phone Number | `'Phone'` |
| Email Address | `'Emailaddress'` |
| Credit Card | `'Creditcard'` |
| Gender | `'Gender'` |
| Object | `'Map'` |
| Array | `'Array'` |
| IP Address | `'Ipaddress'` |
| URL | `'Url'` |
| HTTP Code | `'Httpcodes'` |
| Zip Code | `'Zipcode'` |
| State | `'State'` |
| Date / Time | `'Datetime'` |

# Data Type Validation Patterns

**Contents:**

- *Credit Card*
- *Email Address*
- *Gender*
- *Phone Number*
- *Social Security Number*
- *State*
- *Zip Code*

The following Trifacta® data types utilize regular expressions to validate data against the type. For other data types, validation is performed without using regular expressions.

Patterns matching the following regular expressions are considered valid for the listed data type.

## Credit Card

```
^4\\d{3}[ -]?(\\d{4}[ -]?){3}$
^5[1-5]\\d{2}[ -]?(\\d{4}[ -]?){3}$
^3[4,7]\\d{2}[ -]?\\d{6}[ -]?\\d{5}$
^6(011|(5\\d{2}))[ -]?(\\d{4}[ -]?){3}$
^((35\\d{2}[ -]?(\\d{4}[ -]?){3})|((1800|2131)[ -]?(\\d{4}[ -]?){2}\\d{3}))$
^3((0[0-5]\\d{1})|([6,8]\\d{2}))[ -]?\\d{6}[ -]?\\d{5}$
```

## Email Address

```
^[a-z0-9.!#$%&'*+/=?^_`{|}~-]+@[a-z0-9.-]+(?:\\.[a-z0-9-]+)*\\.[a-z]{2,}$
```

## Gender

```
^(m(?:ale)?|f(?:emale)?)$
```

## Phone Number

```
^(?:(?:\\+?1\\s*(?:[\\.-]\\s*)?)?(?:\\(\\s*(?:[2-9]1[02-9]|[2-9][02-8][0-9])\\s*\\))|(?:[2-9]1[02-9]|[2-9][02-8]1|[2-9][02-8][02-9]))\\s*(?:[.-]\\s*)?)(?:[2-9]1[02-9]|[2-9][02-9]1|[2-9][02-9]{2})\\s*(?:[.-]\\s*)?(?:[0-9]{4})(?:\\s*(?:#|x\\.?|ext\\.?|extension)\\s*(?:\\d+))?$
```

## Social Security Number

```
^(00[1-9]|0[1-9]\\d|66[0-57-9]|6[0-57-9]\\d|[1-57-8]\\d{2})([ -]?)(0[1-9]|[1-9]\\d)\\2(000[1-9]|00[1-9]\\d|0[1-9]\\d{2}|[1-9]\\d{3})$
```

## State

```
^
(al|ak|az|ar|ca|co|ct|de|fl|ga|hi|id|il|in|ia|ks|ky|la|me|md|ma|mi|mn|ms|mo|mt|ne|nv|nh|nj|nm|ny|nc|nd|oh|ok|o
r|pa|ri|sc|sd|tn|tx|ut|vt|va|wa|wv|wi|wy|pr|dc|vi)$
^
(alabama|alaska|arizona|arkansas|california|colorado|connecticut|delaware|florida|georgia|hawaii|idaho|illinoi
s|indiana|iowa|kansas|kentucky|louisiana|maine|maryland|massachusetts|michigan|minnesota|mississippi|missouri|
montana|nebraska|nevada|new hampshire|new jersey|new mexico|new york|north carolina|north
dakota|ohio|oklahoma|oregon|pennsylvania|rhode island|south carolina|south
dakota|tennessee|texas|utah|vermont|virginia|washington|west virginia|wisconsin|wyoming|puerto rico|district
of columbia|virgin islands)$
```

## Zip Code

```
^([\\d]{5}|[\\d]{9})$
^[\\d]{5}-[\\d]{4}
```

# Structure of a URL

This section provides information about the structure of a URL.

A valid value for the URL data type can be composed of the following parts.

Example URL:

```
http://www.app.example.co.uk/support
```

> **NOTE:** IP addresses that include the protocol identifier (`http://1.2.3.4`) do not contain domain identifiers and need to be processed using a different set of methods. It might be easier to remove the protocol identifiers and change the data type to IP Address.

The hierarchy of domain names extends from right to left.

| Element Name | Examples | Wrangle Function | Notes |
|---|---|---|---|
| Top-level domain | • co.uk<br>• com, net, org | *SUFFIX Function* | Every valid URL must have at least one top-level domain.<br><br>> **NOTE:** When the `DOMAIN` function parses a multi-tiered top-level domain such as `co.uk`, the output is the first part of the domain value (e.g. `co`). |
| Second-level domain | example<br><br>app.example | *DOMAIN Function* | This value can be extracted from a valid URL using the `DOMAIN` function. See *DOMAIN Function*. |
| Third-level domain | www | *SUBDOMAIN Function* | This value can be extracted from a valid URL using the `SUBDOMAIN` function. See *SUBDOMAIN Function*. |
| path | /support | | |
| protocol identifier | http://<br><br>https:// | | You can use pattern matching to locate these protocol identifiers. In your Wrangle transforms, use the following Pattern :<br><br>`` `http%?://` ``<br><br>For an example, see *IPTOINT Function*. |
| host | www.app. example. com | *HOST Function* | Protocol identifier (e.g. `http://`) is not included |

# Language Documentation Syntax Notes

In this documentation, working examples are complete  Wrangle  commands. Example:

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | (2 + 3) |
| Parameter: New column name | 'five' |

All syntax commands use the following generalized structure:

| Transformation Name | [Search Term in Transform Builder] |
|---|---|
| Parameter: param1 | param_value1 |
| Parameter: param2 | FUNCTION_NAME(arg1,[arg2]) |
| Parameter: param3 | (expression) |
| Parameter: param4 | 'param_value4' |

- Inputs:
  - For transformations, inputs are called **parameters**.
  - For functions, inputs are called **arguments**.
- Function names are written in ALL CAPS.
- Square brackets [like this] indicate optional parameters or arguments.
- Parentheses are used:
  - To bracket function inputs
  - To bracket expressions, such as `myCol >= 4`. In some of these instances, parentheses may be optional.
- A vertical pipe (|) may be used between parameters or arguments that are paired. Typically, one of them may be used, but not both.

# Source Metadata References

**Contents:**

Wrangle supports a set of variables, which can be used to programmatically reference aspects of the dataset or its source data. These **metadata references** allow you to create individual transformations of much greater scope and flexibility.

> **Tip:** Some transformation steps make access to metadata about the original data source impossible to retain. It's best to use these references, where possible, early in your recipe. Additional information is available below.

> **Tip:** You can use the `$filepath` and `$sourcerownumber` to create a primary key to identify source information for any row in your file-based datasets.

## $filepath

This reference retrieves the fully qualified path for a row of data sourced from a file. As you are working with a dataset in the application, it can be helpful to know where the file from which each row of data originated. Using the `$filepath` function, you can generate columns of data early in your recipe to retain this useful information.

The following transforms might make file path information invalid or otherwise unavailable:

- `pivot`
- `join`
- `unnest`
- `deduplicate`

> **NOTE:** This reference returns null values for values from relational database sources.

> **NOTE:** This reference returns the file path. It does not include the scheme or authority information from the URI. So, protocol identifiers such as `http://` are not available in the output.

### Supported File Formats

**Base file formats:**

| File format | Supported? | Notes |
|---|---|---|
| CSV | Yes | |

| | | |
|---|---|---|
| JSON | Yes | |
| Excel | Limited | Full path to the source location of the Excel file.<br><br>• For uploaded files, this path is to the location in the base storage layer.<br>• If the file contains multiple worksheets, this value includes sheet names. Example: `/path/to/my/Excel/file.xlsx/Sheet1` |
| Compressed files (Gzip, Bzip2, etc) | Limited | Support for single-file archives only. Full path is returned only if the archive contains a single file. |
| folders | Yes | Full path to the file is returned. You can modify the output column to return the folder path only. |

**Additional file formats:**

| File format | Supported? | Notes |
|---|---|---|
| Avro | Yes | |
| Parquet | Yes | |

**Other source types:**

| Format | Supported? | Notes |
|---|---|---|
| Relational tables | No | |

**Limitations**

- The FILEPATH function produces a null value for any samples collected before the feature was enabled, since the information was not available. To see that lineage information, you must switch to the initial sample or collect a new sample.
- After this function is enabled, non-initial samples collected in the future are slightly smaller in size, due to the space consumed by the filepath lineage information that is tracked as part of the sample. You may see a change in the number of rows in your sample.

**Example 1 - Generate filename column**

The following example generates a column containing the filepath information for each row in the dataset:

| | |
|---|---|
| **Transformation Name** | `New formula` |
| **Parameter: Formula type** | `Single row formula` |
| **Parameter: Formula** | `$filepath` |
| **Parameter: New column name** | `'src_filepath'` |

You can use the following additional steps to extract the filename from the above `src_filepath` column:

| | |
|---|---|
| **Transformation Name** | `New formula` |
| **Parameter: Formula type** | `Single row formula` |
| **Parameter: Formula** | `RIGHTFIND(src_filepath, '\/', false, 0)` |
| **Parameter: New column name** | `rightfind_src_filepath` |

| | |
|---|---|
| **Transformation Name** | `New formula` |

| Parameter: Formula type | Single row formula |
|---|---|
| Parameter: Formula | SUBSTRING(src_filepath, rightfind_src_filepath + 1, LEN (src_filepath)) |
| Parameter: New column name | filename |

| Transformation Name | Delete columns |
|---|---|
| Parameter: Columns | rightfind_src_filepath |
| Parameter: Action | Delete selected columns |

### Example 2 - Source row number across dataset with parameters

When you import a dataset with parameters, the `$sourcerownumber` value returns a continuously incrementing row number across all files in the dataset, effectively creating a primary key. Using the following example, you can create a new column to capture the source row number within individual files.

**Source:**

Here is some example data spread across three files after import using a single dataset with parameters.

| column2 | column3 |
|---|---|
| line1-col1 | line1-col2 |
| line2-col1 | line2-col2 |
| line3-col1 | line3-col2 |
| line1-col1 | line1-col2 |
| line2-col1 | line2-col2 |
| line3-col1 | line3-col2 |
| line1-col1 | line1-col2 |
| line2-col1 | line2-col2 |
| line3-col1 | line3-col2 |

As you can see, lineage is hard to determine across the files.

**Transformation:**

Gather the filepath and source row number information into two new columns:

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | $filepath |
| Parameter: New column name | 'filepath' |

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |

| Parameter: Formula | `$sourcerownumber` |
|---|---|
| Parameter: New column name | `'source_row_number'` |

Create a new column called `start_of_file_offset` which contains the offset value of the row from the first row in the file. In the first statement, mark the value of `$sourcerownumber` for the first row of the file, leaving the other rows for the file empty:

| Transformation Name | `New formula` |
|---|---|
| Parameter: Formula type | `Multiple row formula` |
| Parameter: Formula | `IF(PREV($filepath, 1) == $filepath, NULL(), $sourcerownumber)` |
| Parameter: Sort rows by | `$sourcerownumber` |
| Parameter: New column name | `'start_of_file_offset'` |

Create a new column that contains the values from the previous column, with the empty rows filled in with the last previous value, which is the `$sourcerownumber` for the first row of the current file:

| Transformation Name | `New formula` |
|---|---|
| Parameter: Formula type | `Multiple row formula` |
| Parameter: Formula | `FILL(start_of_file_offset, -1, 0)` |
| Parameter: Sort rows by | `$sourcerownumber` |
| Parameter: New column name | `'filled_start_file_offset'` |

Create a new column computing the file-based source row number as the difference between the raw source row number and the start of file offset value computed in the previous step. This generated value is the source row number for a row within its own file:

| Transformation Name | `New formula` |
|---|---|
| Parameter: Formula type | `Single row formula` |
| Parameter: Formula | `($sourcerownumber - filled_start_file_offset) + 1` |
| Parameter: New column name | `'source_row_number_per_file'` |

Delete the columns used for the intermediate calculations:

| Transformation Name | `Delete columns` |
|---|---|
| Parameter: Columns | `filled_start_file_offset,start_of_file_offset` |
| Parameter: Action | `Delete selected columns` |

**Results:**

| column2 | column3 | filepath | source_row_number | source_row_number_per_file |
|---|---|---|---|---|
|  |  |  |  |  |

| | | | | |
|---|---|---|---|---|
| line1-col1 | line1-col2 | /myPath/file001.txt | 1 | 1 |
| line2-col1 | line2-col2 | /myPath/file001.txt | 2 | 2 |
| line3-col1 | line3-col2 | /myPath/file001.txt | 3 | 3 |
| line1-col1 | line1-col2 | /myPath/file002.txt | 4 | 1 |
| line2-col1 | line2-col2 | /myPath/file002.txt | 5 | 2 |
| line3-col1 | line3-col2 | /myPath/file002.txt | 6 | 3 |
| line1-col1 | line1-col2 | /myPath/file003.txt | 7 | 1 |
| line2-col1 | line2-col2 | /myPath/file003.txt | 8 | 2 |
| line3-col1 | line3-col2 | /myPath/file003.txt | 9 | 3 |

## $sourcerownumber

The `$sourcerownumber` variable is a reference to the row number in which the current row originally appeared in the source of the data.

> **Tip:** If the source row information is still available, you can hover over the left side of a row in the data grid to see the source row number in the original source data.

**Limitations:**

- The following transforms might make original row information invalid or otherwise unavailable. In these cases, the reference returns null values:
  - `pivot`
  - `flatten`
  - `join`
  - `lookup`
  - `union`
  - `unnest`
  - `unpivot`

- This reference does not apply to relational database sources.

- For files converted on import in the backend datastore, such as Microsoft Excel, this reference returns the source row value for the converted file on the backend infrastructure. If the conversion results in multiple files, the row numbers are continued across files.

**Example:**

The following example generates a new column containing the source row number for each row in the dataset, if available:

| | |
|---|---|
| **Transformation Name** | `New formula` |
| **Parameter: Formula type** | `Single row formula` |
| **Parameter: Formula** | `$sourcerownumber` |
| **Parameter: New column name** | `'src_rownumber'` |

If you have already used the `$filepath` reference, as in the previous example, you can combine these two columns to create a unique key to the source of each row:

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | MERGE([src_filename,src_rownumber],'-') |
| Parameter: New column name | 'src_key' |

## $col

The `$col` variable is a reference to the column that is currently being evaluated. This variable references the state of the current dataset, instead of the original source.

> **NOTE:** This reference works only for the `edit with formula` transformation (`set` transform).

In the following example, all columns in the dataset that are of String type are converted to uppercase:

| Transformation Name | Edit column with formula |
|---|---|
| Parameter: Columns | All |
| Parameter: Formula | IF(ISMISMATCHED($col, ['String']), $col, UPPER($col)) |

In the above, the wildcard applies the edit to each column. Each column is tested to see if it is mismatched with the String data type. If mismatched, the value in the column (`$col`) is written. Otherwise, the value in the column is converted to uppercase (`UPPER($col)`).

> **Tip:** `$col` is useful for multi-column transformations.

## Flow Parameters

You can create flow parameters that can be referenced in your recipe steps. In your step, you insert the parameter reference token such as the following:

```
${MyParameterName}
```

When the job is executed, this parameter reference is replaced with the corresponding value for it, which can be the default value or an override specified for the flow or the job. Flow parameters are specified through Flow View. For more information, see *Manage Parameters Dialog*.

# Column Reference Syntax

**Contents:**

Wrangle enables you to specify sets of columns using discrete values, ranges, and wildcards. This section describes the syntax associated with these various types of references.

Column references can fit into the following categories:

| Category | Description |
|---|---|
| Single | A reference to a single column. |
| Multiple | References to multiple discrete columns. |
| All | A single reference to all columns in the dataset. |
| Range | References to a set of consecutive columns in the dataset. |
| Advanced | Any of the above categories or combinations of them. |

The sections below describe how to specify the above categories of column references in raw Wrangle .

## Single and Multiple Columns

You can specify single and multiple columns by inserting discrete references to the column name.

**Single column:**

Insert the column name in the Columns textbox:

```
myColumn
```

Example transformation:

| Transformation Name | `Move Columns` |
|---|---|
| **Parameter: Column(s)** | `Multiple` |
| **Parameter: Column** | `myColumn` |
| **Parameter: Option** | `Before` |
| **Parameter: Column** | `myFirstColumn` |

**Multiple columns:**

You can reference multiple discrete columns using comma-separated values:

```
myColumn, myOtherColumn
```

Example transformation:

| Transformation Name | Move Columns |
|---|---|
| Parameter: Column(s) | Multiple |
| Parameter: Column | myColumn,myOtherColumn |
| Parameter: Option | Before |
| Parameter: Column | myFirstColumn |

## All Columns

If needed, you can specify all columns in the dataset using a wildcard. The asterisk character (*) is used to indicate all columns in the dataset:

```
*
```

Example transformation:

| Transformation Name | Edit column with formula |
|---|---|
| Parameter: Column(s) | Advanced |
| Parameter: Column | * |
| Parameter: Formula | set col: * value: average(myCol) |

The above transformation sets the values for all columns to be the AVERAGE value of the `myCol` column.

## Column Ranges

You can use the tilde character (~) to express a range of columns between the start column and the end column, inclusive:

```
myStartColumn~myEndColumn
```

> **NOTE:** If a transformation step is inserted before this one in which the location of one of the columns in the range is changed, then the columns represented by the specified range changes. If the column is no longer present, then this transformation step must be fixed.

Example transformation:

| Transformation Name | Move Columns |
|---|---|
| Parameter: Column(s) | Advanced |
| Parameter: Column | myStartColumn~myEndColumn |
| Parameter: Option | Before |
| Parameter: Column | myFirstColumn |

## Advanced Column References

You can use advanced column references to express combinations of the above types of column reference categories.

```
myStartColumn~myEndColumn, thisColumn2, thisColumn3
```

The above example references:

- The range of columns between myStartColumn and myEndColumn, inclusive
- The thisColumn2 column
- The thisColumn3 column

Example transformation:

| Transformation Name | Edit column with formula |
|---|---|
| Parameter: Column(s) | Advanced |
| Parameter: Column | myStartColumn~myEndColumn, thisColumn2, thisColumn3 |
| Parameter: Formula | POW($col,2) |

For more information on the `$col` reference, see below.

## Column Variable References

When you are applying a transformation step to multiple columns, you cannot reference each column as a parameter in any function in the transformation. Instead, you can insert a variable reference into the function. Below is an example column variable reference used as the input parameter for the SUM function:

```
SUM($col)
```

As the transformation is applied to each column in your column set, the `$col` reference is replaced with the name of the column.

For more information, see *Source Metadata References*.

# Language Index

This section contains an index to functions available in Wrangle (a domain-specific language for data transformation).

## *Aggregate Functions*

| Item | Description |
|---|---|
| *ANY Function* | Extracts a non-null and non-missing value from a specified column. If all values are missing or null, the function returns a null value. |
| *ANYIF Function* | Selects a single non-null value from rows in each group that meet a specific condition. |
| *APPROXIMATEMEDIAN Function* | Computes the approximate median from all row values in a column or group. Input column can be of Integer or Decimal. |
| *APPROXIMATEPERCENTILE Function* | Computes an approximation for a specified percentile across all row values in a column or group. Input column can be of Integer or Decimal. |
| *APPROXIMATEQUARTILE Function* | Computes an approximation for a specified quartile across all row values in a column or group. Input column can be of Integer or Decimal. |
| *AVERAGE Function* | Computes the average (mean) from all row values in a column or group. Input column can be of Integer or Decimal. |
| *AVERAGEIF Function* | Generates the average value of rows in each group that meet a specific condition. Generated value is of Decimal type. |
| *CORREL Function* | Computes the correlation coefficient between two columns. Source values can be of Integer or Decimal type. |
| *COUNTA Function* | Generates the count of non-null rows in a specified column, optionally counted by group. Generated value is of Integer type. |
| *COUNTAIF Function* | Generates the count of non-null values for rows in each group that meet a specific condition. |
| *COUNTDISTINCT Function* | Generates the count of distinct values in a specified column, optionally counted by group. Generated value is of Integer type. |
| *COUNTDISTINCTIF Function* | Generates the count of distinct non-null values for rows in each group that meet a specific condition. |
| *COUNT Function* | Generates the count of rows in the dataset. Generated value is of Integer type. |
| *COUNTIF Function* | Generates the count of rows in each group that meet a specific condition. Generated value is of Integer type. |
| *COVAR Function* | Computes the covariance between two columns using the population method. Source values can be of Integer or Decimal type. |
| *COVARSAMP Function* | Computes the covariance between two columns using the sample method. Source values can be of Integer or Decimal type. |
| *KTHLARGEST Function* | Extracts the ranked value from the values in a column, where $k=1$ returns the maximum value. The value for $k$ must be between 1 and 1000, inclusive. Inputs can be Integer, Decimal, or Datetime. |
| *KTHLARGESTIF Function* | Extracts the ranked value from the values in a column, where $k=1$ returns the maximum value, when a specified condition is met. The value for $k$ must be between 1 and 1000, inclusive. Inputs can be Integer, Decimal, or Datetime. |
| *KTHLARGESTUNIQUE Function* | Extracts the ranked unique value from the values in a column, where $k=1$ returns the maximum value. The value for $k$ must be between 1 and 1000, inclusive. Inputs can be Integer, Decimal, or Datetime. |
| *KTHLARGESTUNIQUEIF Function* | Extracts the ranked unique value from the values in a column, where $k=1$ returns the maximum value, when a specified condition is met. The value for $k$ must be between 1 and 1000, inclusive. Inputs can be Integer, Decimal, or Datetime. |
| *LIST Function* | Extracts the set of values from a column into an array stored in a new column. This function is typically part of an aggregation. |
| *LISTIF Function* | Returns list of all values in a column for rows that match a specified condition. |

| MAX Function | Computes the maximum value found in all row values in a column. Inputs can be Integer, Decimal, or Datetime. |
|---|---|
| MAXIF Function | Generates the maximum value of rows in each group that meet a specific condition. Inputs can be Integer, Decimal, or Datetime. |
| MEDIAN Function | Computes the median from all row values in a column or group. Input column can be of Integer or Decimal. |
| MIN Function | Computes the minimum value found in all row values in a column. Input column can be of Integer, Decimal or Datetime. |
| MINIF Function | Generates the minimum value of rows in each group that meet a specific condition. Inputs can be Integer, Decimal, or Datetime. |
| MODE Function | Computes the mode (most frequent value) from all row values in a column, according to their grouping. Input column can be of Integer, Decimal, or Datetime type. |
| MODEIF Function | Computes the mode (most frequent value) from all row values in a column, according to their grouping. Input column can be of Integer, Decimal, or Datetime type. |
| PERCENTILE Function | Computes a specified percentile across all row values in a column or group. Input column can be of Integer or Decimal. |
| QUARTILE Function | Computes a specified quartile across all row values in a column or group. Input column can be of Integer or Decimal. |
| STDEV Function | Computes the standard deviation across all column values of Integer or Decimal type. |
| STDEVIF Function | Generates the standard deviation of values by group in a column that meet a specific condition. |
| STDEVSAMP Function | Computes the standard deviation across column values of Integer or Decimal type using the sample statistical method. |
| STDEVSAMPIF Function | Generates the standard deviation of values by group in a column that meet a specific condition using the sample statistical method. |
| SUM Function | Computes the sum of all values found in all row values in a column. Input column can be of Integer or Decimal. |
| SUMIF Function | Generates the sum of rows in each group that meet a specific condition. |
| UNIQUE Function | Extracts the set of unique values from a column into an array stored in a new column. This function is typically part of an aggregation. |
| VAR Function | Computes the variance among all values in a column. Input column can be of Integer or Decimal. If no numeric values are detected in the input column, the function returns $0$. |
| VARIF Function | Generates the variance of values by group in a column that meet a specific condition. |
| VARSAMP Function | Computes the variance among all values in a column using the sample statistical method. Input column can be of Integer or Decimal. If no numeric values are detected in the input column, the function returns $0$. |
| VARSAMPIF Function | Generates the variance of values by group in a column that meet a specific condition using the sample statistical method. |

## Logical Functions

| Item | Description |
|---|---|
| Logical Operators | Logical operators (and, or, not) enable you to logically combine multiple expressions to evaluate a larger, more complex expression whose output is `true` or `false`. |
| AND Function | Returns `true` if both arguments evaluate to `true`. Equivalent to the `&&` operator. |
| OR Function | Returns `true` if either argument evaluates to `true`. Equivalent to the `||` operator. |
| NOT Function | Returns `true` if the argument evaluates to `false`, and vice-versa. Equivalent to the `!` operator. |

## Comparison Functions

| Item | Description |
|---|---|
| | |

| | Comparison operators enable you to compare values in the left-hand side of an expression to the values in the right-hand side of an expression. |
|---|---|
| *Comparison Operators* | |
| *ISEVEN Function* | Returns `true` if the argument is an even value. Argument can be an Integer, a function returning Integers, or a column reference. |
| *ISODD Function* | Returns `true` if the argument is an odd value. Argument can be an Integer, a function returning Integers, or a column reference. |
| *IN Function* | Returns `true` if the first parameter is contained in the array of values in the second parameter. |
| *MATCHES Function* | Returns `true` if a value contains a string or pattern. The value to search can be a string literal, a function returning a string, or a reference to a column of String type. |
| *EQUAL Function* | Returns `true` if the first argument is equal to the second argument. Equivalent to the = operator. |
| *NOTEQUAL Function* | Returns `true` if the first argument is not equal to the second argument. Equivalent to the `<>` or `!=` operator. |
| *GREATERTHAN Function* | Returns `true` if the first argument is greater than but not equal to the second argument. Equivalent to the `>` operator. |
| *GREATERTHANEQUAL Function* | Returns `true` if the first argument is greater than or equal to the second argument. Equivalent to the `>=` operator. |
| *LESSTHAN Function* | Returns `true` if the first argument is less than but not equal to the second argument. Equivalent to the `<` operator. |
| *LESSTHANEQUAL Function* | Returns `true` if the first argument is less than or equal to the second argument. Equivalent to the `<=` operator. |

## Math Functions

| Item | Description |
|---|---|
| *Numeric Operators* | Numeric operators enable you to generate new values based on a computation (e.g. $3 + 4$). |
| *NUMFORMAT Function* | Formats a numeric set of values according to the specified number formatting. Source values can be a literal numeric value, a function returning a numeric value, or reference to a column containing an Integer or Decimal values. |
| *ADD Function* | Returns the value of summing the first argument and the second argument. Equivalent to the + operator. |
| *SUBTRACT Function* | Returns the value of subtracting the second argument from the first argument. Equivalent to the − operator. |
| *MULTIPLY Function* | Returns the value of multiplying the first argument by the second argument. Equivalent to the * operator. |
| *DIVIDE Function* | Returns the value of dividing the first argument by the second argument. Equivalent to the / operator. |
| *MOD Function* | Returns the modulo value, which is the remainder of dividing the first argument by the second argument. Equivalent to the % operator. |
| *NEGATE Function* | Returns the opposite of the value that is the first argument. Equivalent to the − operator placed in front of the argument. |
| *SIGN Function* | Computes the positive or negative sign of a given numeric value. The value can be a Decimal or Integer literal, a function returning Decimal or Integer, or a reference to a column containing numeric values. |
| *LCM Function* | Returns the least common multiple shared by the first and second arguments. |
| *ABS Function* | Computes the absolute value of a given numeric value. The value can be a Decimal or Integer literal or a reference to a column containing numeric values. |
| *EXP Function* | Computes the value of *e* raised to the specified power. The value can be a Decimal or Integer literal or a reference to a column containing numeric values. |
| | Computes the logarithm of the first argument with a base of the second argument. |

| | |
|---|---|
| *LOG Function* | |
| *LN Function* | Computes the natural logarithm of an input value. The value can be a Decimal or Integer literal or a reference to a column containing numeric values. |
| *POW Function* | Computes the value of the first argument raised to the value of the second argument. |
| *SQRT Function* | Computes the square root of the input parameter. Input value can be a Decimal or Integer literal or a reference to a column containing numeric values. All generated values are non-negative. |
| *CEILING Function* | Computes the **ceiling** of a value, which is the smallest integer that is greater than the input value. Input can be an Integer, a Decimal, a column reference, or an expression. |
| *FLOOR Function* | Computes the largest integer that is not more than the input value. Input can be an Integer, a Decimal, a column reference, or an expression. |
| *ROUND Function* | Rounds input value to the nearest integer. Input can be an Integer, a Decimal, a column reference, or an expression. Optional second argument can be used to specify the number of digits to which to round. |
| *TRUNC Function* | Removes all digits to the right of the decimal point for any value. Optionally, you can specify the number of digits to which to round. Input can be an Integer, a Decimal, a column reference, or an expression. |
| *NUMVALUE Function* | Converts a string formatted as a number into an Integer or Decimal value by parsing out the specified decimal and group separators. A string or a function returning formatted numbers of String type or a column containing formatted numbers of string type can be inputs. |

## Trigonometry Functions

| Item | Description |
|---|---|
| *SIN Function* | Computes the sine of an input value for an angle measured in radians. The value can be a Decimal or Integer literal or a reference to a column containing numeric values. |
| *COS Function* | Computes the cosine of an input value for an angle measured in radians. The value can be a Decimal or Integer literal or a reference to a column containing numeric values. |
| *TAN Function* | Computes the tangent of an input value for an angle measured in radians. The value can be a Decimal or Integer literal or a reference to a column containing numeric values. |
| *ASIN Function* | For input values between -1 and 1 inclusive, this function returns the angle in radians whose sine value is the input. This function is the inverse of the sine function. The value can be a Decimal or Integer literal or a reference to a column containing numeric values. |
| *ACOS Function* | For input values between -1 and 1 inclusive, this function returns the angle in radians whose cosine value is the input. This function is the inverse of the cosine function. The value can be a Decimal or Integer literal or a reference to a column containing numeric values. |
| *ATAN Function* | For input values between -1 and 1 inclusive, this function returns the angle in radians whose tangent value is the input. This function is the inverse of the tangent function. The value can be a Decimal or Integer literal or a reference to a column containing numeric values. |
| *SINH Function* | Computes the hyperbolic sine of an input value for a hyperbolic angle measured in radians. The value can be a Decimal or Integer literal or a reference to a column containing numeric values. |
| *COSH Function* | Computes the hyperbolic cosine of an input value for a hyperbolic angle measured in radians. The value can be a Decimal or Integer literal or a reference to a column containing numeric values. |
| *TANH Function* | Computes the hyperbolic tangent of an input value for a hyperbolic angle measured in radians. The value can be a Decimal or Integer literal or a reference to a column containing numeric values. |
| *ASINH Function* | Computes the arcsine of an input value for a hyperbolic angle measured in radians. The value can be a Decimal or Integer literal or a reference to a column containing numeric values. |
| *ACOSH Function* | Computes the arccosine of an input value for a hyperbolic angle measured in radians. The value can be a Decimal or Integer literal or a reference to a column containing numeric values. |
| *ATANH Function* | Computes the arctangent of an input value for a hyperbolic angle measured in radians. The value can be a Decimal or Integer literal or a reference to a column containing numeric values. |
| *DEGREES Function* | Computes the degrees of an input value measuring the radians of an angle. The value can be a Decimal or Integer literal or a reference to a column containing numeric values. |

| Item | Description |
|---|---|
| | Computes the radians of an input value measuring degrees of an angle. The value can be a Decimal or Integer literal or a reference to a column containing numeric values. |

*RADIANS Function* appears in the left cell above.

## Date Functions

| Item | Description |
|---|---|
| *DATE Function* | Generates a date value from three inputs of Integer type: year, month, and day. |
| *TIME Function* | Generates time values from three inputs of Integer type: hour, minute, and second. |
| *DATETIME Function* | Generates a Datetime value from the following inputs of Integer type: year, month, day, hour, minute, and second. |
| *DATEADD Function* | Add a specified number of units to a valid date. Units can be any supported Datetime unit (e.g. `minute, month, year`, etc.). Input must be a column reference containing dates. |
| *DATEDIF Function* | Calculates the difference between two valid date values for the specified units of measure. |
| *DATEFORMAT Function* | Formats a specified Datetime set of values according to the specified date format. Source values can be a reference to a column containing Datetime values. |
| *UNIXTIMEFORMAT Function* | Formats a set of Unix timestamps according to a specified date formatting string. |
| *MONTH Function* | Derives the month integer value from a Datetime value. Source value can be a a reference to a column containing Datetime values or a literal. |
| *MONTHNAME Function* | Derives the full name from a Datetime value of the corresponding month as a String. Source value can be a reference to a column containing Datetime values or a literal. |
| *EOMONTH Function* | Returns the serial date number for the last day of the month before or after a specified number of months from a starting date. |
| *YEAR Function* | Derives the four-digit year value from a Datetime value. Source value can be a a reference to a column containing Datetime values or a literal. |
| *DAY Function* | Derives the numeric day value from a Datetime value. Source value can be a a reference to a column containing Datetime values or a literal. |
| *WEEKNUM Function* | Derives the numeric value for the week within the year (`1`, `2`, etc.). Input must be the output of the `DATE` function or a reference to a column containing Datetime values. The output of this function increments on Sunday. |
| *WEEKDAY Function* | Derives the numeric value for the day of the week (`1`, `2`, etc.). Input must be a reference to a column containing Datetime values. |
| *WEEKDAYNAME Function* | Derives the full name from a Datetime value of the corresponding weekday as a String. Source value can be a reference to a column containing Datetime values or a literal. |
| *HOUR Function* | Derives the hour value from a Datetime value. Generated hours are expressed according to the 24-hour clock. |
| *MINUTE Function* | Derives the minutes value from a Datetime value. Minutes are expressed as integers from 0 to 59. |
| *SECOND Function* | Derives the seconds value from a Datetime value. Source value can be a a reference to a column containing Datetime values or a literal. |
| *UNIXTIME Function* | Derives the Unixtime (or epoch time) value from a Datetime value. Source value can be a reference to a column containing Datetime values. |
| *NOW Function* | Derives the timestamp for the current time in UTC time zone. You can specify a different time zone by optional parameter. |
| *TODAY Function* | Derives the value for the current date in UTC time zone. You can specify a different time zone by optional parameter. |
| *PARSEDATE Function* | Evaluates an input against the default input formats or (if specified) an array of Datetime format strings in their listed order. If the input matches one of the formats, the function outputs a Datetime value. |
| *NETWORKDAYS Function* | Calculates the number of working days between two specified dates, assuming Monday - Friday workweek. Optional list of holidays can be specified. |
| *NETWORKDAYSINTL Function* | Calculates the number of working days between two specified dates. Optionally, you can specify which days of the week are working days as an input parameter. Optional list of holidays can be specified. |
| *MINDATE Function* | Computes the minimum value found in all row values in a Datetime column. |

| | |
|---|---|
| *MAXDATE Function* | Computes the maximum value found in all row values in a Datetime column. |
| *MODEDATE Function* | Computes the most frequent (mode) value found in all row values in a Datetime column. |
| *WORKDAY Function* | Calculates the work date that is before or after a start date, as specified by a number of days. A set of holiday dates can be optionally specified. |
| *WORKDAYINTL Function* | Calculates the work date that is before or after a start date, as specified by a number of days. You can also specify which days of the week are working days and a list of holidays via parameters. |
| *CONVERTFROMU TC Function* | Converts Datetime value to corresponding value of the specified time zone. Input can be a column of Datetime values, a literal Datetime value, or a function returning Datetime values. |
| *CONVERTTOUTC Function* | Converts Datetime value in specified time zone to corresponding value in UTC time zone. Input can be a column of Datetime values, a literal Datetime value, or a function returning Datetime values. |
| *CONVERTTIMEZO NE Function* | Converts Datetime value in specified time zone to corresponding value second specified time zone. Input can be a column of Datetime values, a literal Datetime value, or a function returning Datetime values. |
| *MINDATEIF Function* | Returns the minimum Datetime value of rows in each group that meet a specific condition. Set of values must valid Datetime values. |
| *MAXDATEIF Function* | Returns the maximum Datetime value of rows in each group that meet a specific condition. Set of values must valid Datetime values. |
| *MODEDATEIF Function* | Returns the most common Datetime value of rows in each group that meet a specific condition. Set of values must valid Datetime values. |
| *KTHLARGESTDAT E Function* | Extracts the ranked Datetime value from the values in a column, where $k=1$ returns the maximum value. The value for $k$ must be between 1 and 1000, inclusive. Inputs must be valid Datetime values. |
| *KTHLARGESTUNI QUEDATE Function* | Extracts the ranked unique Datetime value from the values in a column, where $k=1$ returns the maximum value. The value for $k$ must be between 1 and 1000, inclusive. Inputs must be Datetime. |
| *KTHLARGESTUNI QUEDATEIF Function* | Extracts the ranked unique Datetime value from the values in a column, where $k=1$ returns the maximum value, when a specified condition is met. The value for $k$ must be between 1 and 1000, inclusive. Inputs must be Datetime. |
| *KTHLARGESTDAT EIF Function* | Extracts the ranked Datetime value from the values in a column, where $k=1$ returns the maximum value, when a specified condition is met. The value for $k$ must be between 1 and 1000, inclusive. Inputs must be Datetime. |
| *SERIALNUMBER Function* | Generates a serial date number from a valid date value. |

## String Functions

| Item | Description |
|---|---|
| *CHAR Function* | Generates the Unicode character corresponding to an inputted Integer value. |
| *UNICODE Function* | Generates the Unicode index value for the first character of the input string. |
| *UPPER Function* | All alphabetical characters in the input value are converted to uppercase in the output value. |
| *LOWER Function* | All alphabetical characters in the input value are converted to lowercase in the output value. |
| *PROPER Function* | Converts an input string to propercase. Input can be a column reference or a string literal. |
| *TRIM Function* | Removes leading and trailing whitespace from a string. Spacing between words is not removed. |
| *REMOVEWHITESPAC E Function* | Removes all whitespace from a string, including leading and trailing whitespace and all whitespace within the string. |
| *REMOVESYMBOLS Function* | Removes all characters from a string that are not letters, numbers, accented Latin characters, or whitespace. |
| *LEN Function* | Returns the number of characters in a specified string. String value can be a column reference or string literal. |
| *FIND Function* | Returns the index value in the input string where a specified matching string is located in provided column, string literal, or function returning a string. Search is conducted left-to-right. |
| *RIGHTFIND Function* | Returns the index value in the input string where the last instance of a matching string is located. Search is conducted right-to-left. |

| | |
|---|---|
| *FINDNTH Function* | Returns the position of the nth occurrence of a letter or pattern in the input string where a specified matching string is located in the provided column. You can search either from left or right. |
| *SUBSTRING Function* | Matches some or all of a string, based on the user-defined starting and ending index values within the string. |
| *SUBSTITUTE Function* | Replaces found string literal or pattern or column with a string, column, or function returning strings. |
| *LEFT Function* | Matches the leftmost set of characters in a string, as specified by parameter. The string can be specified as a column reference or a string literal. |
| *RIGHT Function* | Matches the right set of characters in a string, as specified by parameter. The string can be specified as a column reference or a string literal. |
| *PAD Function* | Pads string values to be a specified minimum length by adding a designated character to the left or right end of the string. Returned value is of String type. |
| *MERGE Function* | Merges two or more columns of String type to generate output of String type. Optionally, you can insert a delimiter between the merged values. |
| *STARTSWITH Function* | Returns `true` if the leftmost set of characters of a column of values matches a pattern. The source value can be any data type, and the pattern can be a Pattern , regular expression, or a string. |
| *ENDSWITH Function* | Returns `true` if the rightmost set of characters of a column of values matches a pattern. The source value can be any data type, and the pattern can be a Pattern , regular expression, or a string. |
| *REPEAT Function* | Repeats a string a specified number of times. The string can be specified as a String literal, a function returning a String, or a column reference. |
| *EXACT Function* | Returns `true` if the second string evaluates to be an exact match of the first string. Source values can be string literals, column references, or expressions that evaluate to strings. |
| *STRINGGREATERTHAN Function* | Returns `true` if the first string evaluates to be greater than the second string, based on a set of common collation rules. |
| *STRINGGREATERTHANEQUAL Function* | Returns `true` if the first string evaluates to be greater than or equal to the second string, based on a set of common collation rules. |
| *STRINGLESSTHAN Function* | Returns `true` if the first string evaluates to be less than the second string, based on a set of common collation rules. |
| *STRINGLESSTHANEQUAL Function* | Returns `true` if the first string evaluates to be less than or equal to the second string, based on a set of common collation rules. |
| *DOUBLEMETAPHONE Function* | Returns a two-element array of primary and secondary phonetic encodings for an input string, based on the Double Metaphone algorithm. |
| *DOUBLEMETAPHONE EQUALS Function* | Compares two input strings using the Double Metaphone algorithm. An optional threshold parameter can be modified to adjust the tolerance for matching. |
| *TRANSLITERATE Function* | Transliterates Asian script characters from one script form to another. The string can be specified as a column reference or a string literal. |
| *TRIMQUOTES Function* | Removes leading and trailing quotes or double-quotes from a string. Quote marks in the middle of the string are not removed. |
| *BASE64ENCODE Function* | Converts an input value to base64 encoding with optional padding with an equals sign (=). Input can be of any type. Output type is String. |
| *BASE64DECODE Function* | Converts an input base64 value to text. Output type is String. |

## Nested Functions

| Item | Description |
|---|---|
| *ARRAYCONCAT Function* | Combines the elements of one array with another, listing all elements of the first array before listing all elements of the second array. |
| *ARRAYCROSS Function* | Generates a nested array containing the cross-product of all elements in two or more arrays. |
| *ARRAYELEMENTAT Function* | Computes the 0-based index value for an array element in the specified column, array literal, or function that returns an array. |
| | |

| | Computes the index at which a specified element is first found within an array. Indexing is left to right. |
|---|---|
| *ARRAYINDEXOF Function* | Computes the index at which a specified element is first found within an array. Indexing is left to right. |
| *ARRAYINTERSECT Function* | Generates an array containing all elements that appear in multiple input arrays, referenced as column names or array literals. |
| *ARRAYLEN Function* | Computes the number of elements in the arrays in the specified column, array literal, or function that returns an array. |
| *ARRAYMERGEELE MENTS Function* | Merges the elements of an array in left to right order into a string. Values are optionally delimited by a provided delimiter. |
| *ARRAYRIGHTINDE XOF Function* | Computes the index at which a specified element is first found within an array, when searching right to left. Returned value is based on left-to-right indexing. |
| *ARRAYSLICE Function* | Returns an array containing a slice of the input array, as determined by starting and ending index parameters. |
| *ARRAYSORT Function* | Sorts array values in the specified column, array literal, or function that returns an array in ascending or descending order. |
| *ARRAYSTOMAP Function* | Combines one array containing keys and another array containing values into an Object of key-value pairs. |
| *ARRAYUNIQUE Function* | Generates an array of all unique elements among one or more arrays. |
| *ARRAYZIP Function* | Combines multiple arrays into a single nested array, with element 1 of array 1 paired with element 2 of array 2 and so on. Arrays are expressed as column names or as array literals. |
| *FILTEROBJECT Function* | Filters the keys and values from an Object data type column based on a specified key value. |
| *KEYS Function* | Extracts the key values from an Object data type column and stores them in an array of String values. |
| *LISTAVERAGE Function* | Computes the average of all numeric values found in input array. Input can be an array literal, a column of arrays, or a function returning an array. Input values must be of Integer or Decimal type. |
| *LISTMAX Function* | Computes the maximum of all numeric values found in input array. Input can be an array literal, a column of arrays, or a function returning an array. Input values must be of Integer or Decimal type. |
| *LISTMIN Function* | Computes the minimum of all numeric values found in input array. Input can be an array literal, a column of arrays, or a function returning an array. Input values must be of Integer or Decimal type. |
| *LISTMODE Function* | Computes the most common value of all numeric values found in input array. Input can be an array literal, a column of arrays, or a function returning an array. Input values must be of Integer or Decimal type. |
| *LISTSTDEV Function* | Computes the standard deviation of all numeric values found in input array. Input can be an array literal, a column of arrays, or a function returning an array. Input values must be of Integer or Decimal type. |
| *LISTSUM Function* | Computes the sum of all numeric values found in input array. Input can be an array literal, a column of arrays, or a function returning an array. Input values must be of Integer or Decimal type. |
| *LISTVAR Function* | Computes the variance of all numeric values found in input array. Input can be an array literal, a column of arrays, or a function returning an array. Input values must be of Integer or Decimal type. |

## Type Functions

| Item | Description |
|---|---|
| *NULL Function* | The `NULL` function generates null values. |
| *IFNULL Function* | The `IFNULL` function writes out a specified value if the source value is a null. Otherwise, it writes the source value. Input can be a literal, a column reference, or a function. |
| *IFMISSING Function* | The `IFMISSING` function writes out a specified value if the source value is a null or missing value. Otherwise, it writes the source value. Input can be a literal, a column reference, or a function. |
| *IFMISMATC HED Function* | The `IFMISMATCHED` function writes out a specified value if the input expression does not match the specified data type or typing array. Otherwise, it writes the source value. Input can be a literal, a column reference, or a function. |
| | The `IFVALID` function writes out a specified value if the input expression matches the specified data type. Otherwise, it |

| | |
|---|---|
| *IFVALID Function* | writes the source value. Input can be a literal, a column reference, or a function. |
| *ISNULL Function* | The `ISNULL` function tests whether a column of values contains null values. For input column references, this function returns `true` or `false`. |
| *ISMISSING Function* | The `ISMISSING` function tests whether a column of values is missing or null. For input column references, this function returns `true` or `false`. |
| *ISMISMATC HED Function* | Tests whether a set of values is not valid for a specified data type. |
| *VALID Function* | Tests whether a set of values is valid for a specified data type and is not a null value. |
| *PARSEINT Function* | Evaluates a String input against the Integer datatype. If the input matches, the function outputs an Integer value. Input can be a literal, a column of values, or a function returning String values. |
| *PARSEBOO L Function* | Evaluates a String input against the Boolean datatype. If the input matches, the function outputs a Boolean value. Input can be a literal, a column of values, or a function returning String values. |
| *PARSEFLO AT Function* | Evaluates a String input against the Decimal datatype. If the input matches, the function outputs a Decimal value. Input can be a literal, a column of values, or a function returning String values. |
| *PARSEARR AY Function* | Evaluates a String input against the Array datatype. If the input matches, the function outputs an Array value. Input can be a literal, a column of values, or a function returning String values. |
| *PARSEOBJ ECT Function* | Evaluates a String input against the Object datatype. If the input matches, the function outputs an Object value. Input can be a literal, a column of values, or a function returning String values. |
| *PARSESTRI NG Function* | Evaluates an input against the String datatype. If the input matches, the function outputs a String value. Input can be a literal, a column of values, or a function returning values. Values can be of any data type. |

## Window Functions

| Item | Description |
|---|---|
| *PREV Function* | Extracts the value from a column that is a specified number of rows before the current value. |
| *NEXT Function* | Extracts the value from a column that is a specified number of rows after the current value. |
| *FILL Function* | Fills any missing or null values in the specified column with the most recent non-blank value, as determined by the specified window of rows before and after the blank value. |
| *RANK Function* | Computes the rank of an ordered set of value within groups. Tie values are assigned the same rank, and the next ranking is incremented by the number of tie values. |
| *DENSERANK Function* | Computes the rank of an ordered set of value within groups. Tie values are assigned the same rank, and the next ranking is incremented by 1. |
| *ROLLINGAVERAGE Function* | Computes the rolling average of values forward or backward of the current row within the specified column. |
| *ROLLINGMODE Function* | Computes the rolling mode (most common value) forward or backward of the current row within the specified column. Input values can be Integer, Decimal, or Datetime data type. |
| *ROLLINGMAX Function* | Computes the rolling maximum of values forward or backward of the current row within the specified column. Inputs can be Integer, Decimal, or Datetime. |
| *ROLLINGMIN Function* | Computes the rolling minimum of values forward or backward of the current row within the specified column. Inputs can be Integer, Decimal, or Datetime. |
| *ROLLINGMODEDATE Function* | Computes the rolling mode (most common value) forward or backward of the current row within the specified column. Input values must be of Datetime data type. |
| *ROLLINGMAXDATE Function* | Computes the rolling maximum of date values forward or backward of the current row within the specified column. Inputs must be of Datetime type. |
| *ROLLINGMINDATE Function* | Computes the rolling minimum of Date values forward or backward of the current row within the specified column. Inputs must be of Datetime type. |
| *ROLLINGSUM Function* | Computes the rolling sum of values forward or backward of the current row within the specified column. |

| | |
|---|---|
| *ROLLINGSTDEV Function* | Computes the rolling standard deviation of values forward or backward of the current row within the specified column. |
| *ROLLINGSTDEVSAMP Function* | Computes the rolling standard deviation of values forward or backward of the current row within the specified column using the sample statistical method. |
| *ROLLINGVAR Function* | Computes the rolling variance of values forward or backward of the current row within the specified column. |
| *ROLLINGVARSAMP Function* | Computes the rolling variance of values forward or backward of the current row within the specified column using the sample statistical method. |
| *ROLLINGCOUNTA Function* | Computes the rolling count of non-null values forward or backward of the current row within the specified column. |
| *ROLLINGKTHLARGEST Function* | Computes the rolling *kth* largest value forward or backward of the current row. Inputs can be Integer, Decimal, or Datetime. |
| *ROLLINGKTHLARGEST UNIQUE Function* | Computes the rolling unique *kth* largest value forward or backward of the current row. Inputs can be Integer, Decimal, or Datetime. |
| *ROLLINGLIST Function* | Computes the rolling list of values forward or backward of the current row within the specified column and returns an array of these values. |
| *ROWNUMBER Function* | Generates a new column containing the row number as sorted by the `order` parameter and optionally grouped by the `group` parameter. |
| *SESSION Function* | Generates a new session identifier based on a sorted column of timestamps and a specified rolling timeframe. |

## Other Functions

| Item | Description |
|---|---|
| *COALESCE Function* | Function returns the first non-missing value found in an array of columns. |
| *RAND Function* | The `RAND` function generates a random real number between 0 and 1. The function accepts an optional integer parameter, which causes the same set of random numbers to be generated with each job execution. |
| *RANDBETWEEN Function* | Generates a random integer between a low and a high number. Two inputs may be Integer or Decimal types, functions returning these types, or column references. |
| *PI Function* | The `PI` function generates the value of pi to 15 decimal places: 3.1415926535897932. |
| *SOURCEROWNUMBER Function* | Returns the row number of the current row as it appeared in the original source dataset before any steps had been applied. |
| *IF Function* | The `IF` function allows you to build if/then/else conditional logic within your transforms. |
| *CASE Function* | The `CASE` function allows you to perform multiple conditional tests on a set of expressions within a single statement. When a test evaluates to `true`, a corresponding output is generated. Outputs may be a literal or expression. |
| *Ternary Operators* | Ternary operators allow you to build if/then/else conditional logic within your transforms. Please use the `IF` function instead. |
| *IPTOINT Function* | Computes an integer value for a four-octet internet protocol (IP) address. Source value must be a valid IP address or a column reference to IP addresses. |
| *IPFROMINT Function* | Computes a four-octet internet protocol (IP) address from a 32-bit integer input. |
| *RANGE Function* | Computes an array of integers, from a beginning integer to an end (stop) integer, stepping by a third parameter. |
| *HOST Function* | Finds the host value from a valid URL. Input values must be of URL or String type and can be literals or column references. |
| *DOMAIN Function* | Finds the value for the domain from a valid URL. Input values must be of URL or String type. |
| *SUBDOMAIN Function* | Finds the value a subdomain value from a valid URL. Input values must be of URL or String type. |
| *SUFFIX Function* | Finds the suffix value after the domain from a valid URL. Input values must be of URL or String type. |

| URLPARAMS Function | Extracts the query parameters of a URL into an Object. The Object keys are the parameter's names, and its values are the parameter's values. Input values must be of URL or String type. |
|---|---|

# Language Appendices

This section contains additional topics on  Wrangle .

# Transforms

> **NOTE:** Transforms are a part of the underlying language, which is not directly accessible to users. This content is maintained for reference purposes only. For more information on the user-accessible equivalent to transforms, see *Transformation Reference*.

In  Wrangle , a **transform** is an action applied to your dataset. Each step of your recipe corresponds to a fully specified transform.

> **Tip:** To see transforms by category, click the sort buttons in the Category header in the online documentation.

| Transform | Category | Description |
|-----------|----------|-------------|
| *Case Transform* | Other | Perform if/then/else or case logic on the rows in your dataset. |
| *Comment Transform* | Other | Inserts a non-functional comment as a recipe step. |
| *Countpattern Transform* | Search and Replace | Counts the number of instances of a specified pattern in a column and writes that value into a newly generated column. Source column is unchanged. |
| *Deduplicate Transform* | Manage Rows | Removes exact duplicate rows from your dataset. Duplicate rows are identified by exact matches between values. For example, two strings with different capitalization do not match. |
| *Delete Transform* | Manage Rows | Deletes a set of rows in your dataset, based on a condition specified in the `row` expression. If the conditional expression is `true`, then the row is deleted. |
| *Derive Transform* | Manage Columns | Generate a new column where the values are the output of the `value` expression. Expression can be calculated based on values specified in the `group` parameter. Output column can be named as needed. |
| *Drop Transform* | Manage Columns | Removes the specified column or columns permanently from your dataset. |
| *Extract Transform* | Search and Replace | Extracts a subset of data from one column and inserts it into a new column, based on a specified string or pattern. The source column in unmodified. |
| *Extractkv Transform* | Search and Replace | Extracts key-value pairs from a source column and writes them to a new column. Source column must be of String type, although the data can be formatted as other data types. |
| *Extractlist Transform* | Search and Replace | Extracts a set of values based on a specified pattern from a source column of any data type. The generated column contains an array of occurrences of the specified pattern. While the new column contains array data, the data type of the new column is sometimes inferred as String. |
| *Filter Transform* | Manage Rows | Keep or delete rows in your dataset based on a defined type of filter. |
| *Flatten Transform* | Nested Data | Unpacks array data into separate rows for each value. |
| *Header Transform* | Initial Parsing | Uses one row from the dataset sample as the header row for the table. Each value in this row becomes the name of the column in which it is located. |
| *Keep Transform* | Manage Rows | Retains a set of rows in your dataset, which are specified by the conditional in the `row` expression. All other rows are removed from the dataset. |
| *Merge Transform* | Manage Columns | Merges two or more columns in your dataset to create a new column of String type. Optionally, you can insert a delimiter between the merged values. |
|  | Manage | Moves the specified column or columns before or after another column in your dataset. |

| Move Transform | Columns | |
|---|---|---|
| *Nest Transform* | Nested Data | Creates an Object or Array of values using column names and their values as key-value pairs for one or more columns. Generated column type is determined by the `into` parameter. |
| *Pivot Transform* | Nested Data | The `pivot` transform can be used to aggregate or pivot your data into columns and aggregate the results. Reshape your dataset into summary information. <br><br> When you aggregate data, calculations are performed on column values, which are then grouped and ordered based on specified parameters. <br><br> When you pivot data, the values of a selected column become new columns in the dataset, each of which contains a summary calculation that you specify. This calculation can be based on all rows for totals across the dataset or based on group of rows you define in the transform. |
| *Rename Transform* | Manage Columns | Renames one or more columns to specified names or append or prepend column names with specific values. |
| *Replace Transform* | Search and Replace | Replaces values within the specified column or columns based on the string literal, pattern, or location within the cell value, as specified in the transform. |
| *Set Transform* | Search and Replace | Replaces all values in the specified column with the specified value, which can be a literal or an expression. You can specify an optional `row:` parameter, containing a conditional test to identify the rows where the replacement is to be made within the column. |
| *Settype Transform* | Manage Columns | Sets the data type of the specified column. This transform does not modify the source values. The data in the column is re-inferred against the specified data type, which can change the results of column profiling. |
| *Split Transform* | Initial Parsing | Splits the specified column into separate columns of data based on the delimiters in the transform. Delimiters can be specified in a number of methods described below. |
| *Splitrows Transform* | Initial Parsing | Splits a column of values into separate rows of data based on the specified delimiter. You can split rows only on String literal values. Pattern-based row splitting is not supported. |
| *Unnest Transform* | Nested Data | Unpacks nested data from an Array or Object column to create new rows or columns based on the keys in the source data. This transform works differently on columns of Object or Array type. |
| *Unpivot Transform* | Nested Data | Reshapes the layout of data by merging one or more columns into key and value columns. Keys are the names of input columns, and the values are the cell values from the source columns. Rows of data are duplicated, once for each input column. |
| *Valuestocols Transform* | Manage Columns | For each unique value in a column, a separate column is created. For each row that contains the value in the source column, an indicator value is inserted in the new column. This value can be a literal value or the output of a function. If no indicator value is generated, a null value is written. |
| *Window Transform* | Aggregation | The `window` transform enables you to perform summations and calculations based on a rolling window of data relative to the current row. For example, you can compute the rolling average for a specified column for the current row value and the nine preceding rows. This transform is particularly useful for processing time or otherwise sequential data. |

## Other Transforms:

| Transform | Category | Description |
|---|---|---|
| *Sort Transform* | Manage Rows | Sorts the dataset based on one or more columns in ascending or descending order. You can also sort based on the order of rows when the dataset was created. |

# Case Transform

**Contents:**

---

> **NOTE:** Transforms are a part of the underlying language, which is not directly accessible to users. This content is maintained for reference purposes only. For more information on the user-accessible equivalent to transforms, see *Transformation Reference*.

Performs conditional transformation of data with a single statement using if-then-else logic or with multiple statements using case logic. Results are generated in a new column.

> **NOTE:** If you are running your job on Spark, avoid creating single conditional transformations with deeply nested sets of conditions. On Spark, these jobs can time out, and deeply nested steps can be difficult to debug. Instead, break up your nesting into smaller conditional transformations of multiple steps.

There are function equivalents to this transformation:

- *IF Function*
- *CASE Function*

## Basic Usage

### Example - if/then/else

This example illustrates a single if/then/else construction:

```
case if: testScore >= 60 then: 'yes' else: 'no' as: 'passedTest'
```

**Output:** If a value in the `testScore` is greater than or equal to 60, a value of `yes` is written into the new `passed Test` column. Otherwise, a value of `no` is written.

### Example - Case (single column)

This example shows how to step through a sequence of case tests applied to a single column.

```
case col: custName colCases: ['Big Co',0.2],['Little Guy Ltd',0.05] default: 0 as:
'discountRate'
```

**Output:** Checks names in the `custName` column and writes discount values based on exact matches of values in the column:

| custName value | discountRate |
|---|---|
| Big Co | 0.2 |
| Little Guy Ltd | 0.05 |
| default (if no matches) | 0 |

**Example - Case (custom conditions)**

The following example illustrates how to construct case transforms with multiple independent conditions. Tests can come from arbitrary columns and expressions.

- The first case is tested:
    - If `true`, then the listed value is written to the new column.
    - If `false`, then the next case is tested.
- If none of the stated cases evaluates to `true`, then the default value is written.

```
case cases: [totalOrdersQ3 < 10, true], [lastOrderDays > 60, true] default: false as:
'sendCheckinEmail'
```

**Output:** If the total orders in Q3 < 10 OR the last order was placed more than 60 days ago, then write `true` in the `sendCheckinEmail`. Otherwise, write `false`.

| Logic | Test | SendCheckinEmail |
|---|---|---|
| if | `totalOrdersQ3 < 10` | `true` |
| if above is false | `lastOrderDays > 60` | `true` |
| if above is false | write default | `false` |

## Syntax and Parameters

```
case [if: if_expression] [then:'str_if_true'] [else:'str_if_false] [col:col1] [colCases:
[[Match1,Val1]],[[Match2,Val2]] [cases: [[Exp3,Val3]],[[Exp4,Val4]] [default:default_val]
as: 'new_column_name'
```

| Token | Required? | Data Type | Description |
|---|---|---|---|
| case | Y | transform | Name of the transform |
| if | N | string | (For single if/then/else) Expression that is tested must evaluate to `true` or `false`. |
| then | N | string | (For single if/then/else) Value written to the new column if the if expression is `true`. |
| else | N | string | (For single if/then/else) Value written to the new column if the if expression is `false`. |
| col | N | string | (For single-column case) Name of column whose values are to be tested. |
| colCases | N | comma-separated arrays | (For single-column case) Matrix of string-value pairs:<br><br>• First entry is the value to match.<br>• Second entry is the value written to the new column if a match appears |
| cases | N | comma- | (For custom conditions case) Matrix of expression-value pairs: |

| | | separated arrays | • First entry is the expression to evaluate.<br>• Second entry is the value to write if the expression is `true`. |
|---|---|---|---|
| default | N | any | (For single-column case and custom condition case) If no matches are made, this value is written to the new column. |
| as | Y | string | Name of the new column where results are written. |

For more information on syntax standards, see *Language Documentation Syntax Notes.*

**if**

For if-then-else condition types, this value is an expression to test. Expression must evaluate to `true` or `false`.

**Usage Notes:**

| Required? | Data Type |
|---|---|
| Required for if-the-else condition type | String (expression) |

**then**

For if-then-else condition types, this value is a literal value to write in the output column if the expression evaluates to `true`.

**Usage Notes:**

| Required? | Data Type |
|---|---|
| Required for if-the-else condition type | String or other literal type |

**else**

For if-then-else condition types, this value is a literal value to write in the output column if the expression evaluates to `false`.

**Usage Notes:**

| Required? | Data Type |
|---|---|
| Required for if-the-else condition type | String or other literal type |

**col**

For single-case condition types, this value identifies the column to test.

**Usage Notes:**

| Required? | Data Type |
|---|---|
| Required for single-case condition type | String (column name) |

**colCases**

For single-case condition types, this parameter contains a comma-separated set of two-value arrays.

- **Array value 1:** A literal value to match in the specified column.
- **Array value 2:** If the value is matched, this value is written into the output column.

You can specify one or more cases as comma-separated two-value arrays.

**Usage Notes:**

| Required? | Data Type |
|---|---|
| Required for single-case condition type | Array (comma-separated list) |

### cases

For multi-case condition types, this parameter contains a comma-separated set of two-value arrays.

- **Array value 1:** An expression to test, which must evaluate to `true` or `false`.
- **Array value 2:** If the value is matched, this value is written into the output column.

You can specify one or more cases as comma-separated two-value arrays.

**Usage Notes:**

| Required? | Data Type |
|---|---|
| Required for single-case condition type | Array (comma-separated list) |

### default

For single-case and multi-case condition types, this parameter defines the value to write in the new column if none of the cases yields a `true` result.

**Usage Notes:**

| Required? | Data Type |
|---|---|
| Required for single-case condition type | Literal of any data type |

### as

Name of the new column that is being generated. If the `as` parameter is not specified, a default name is used.

**Usage Notes:**

| Required? | Data Type |
|---|---|
| Yes | String (column name) |

## Examples

> **Tip:** For additional examples, see *Common Tasks*.

See above.

# Comment Transform

> **NOTE:** Transforms are a part of the underlying language, which is not directly accessible to users. This content is maintained for reference purposes only. For more information on the user-accessible equivalent to transforms, see *Transformation Reference*.

Inserts a non-functional comment as a recipe step. Commented steps are ignored during job execution.

Multi-line comments are not supported.

## Basic Usage

Depending on how the comment is inserted, the format of the step may vary.

**Manually inserted comments:**

In the Search panel, enter the following text in the Transformation textbox:

```
comment
```

Then, specify a value in the comment textbox.

**Pasted comments:**

Text values of the following formats can be pasted into the Transformation textbox. These steps are converted to comment transforms:

```
// This is a comment.
```

```
/* This is also a comment. */
```

**Output:** Output is unaffected by the comment. Comments are highlighted in a different color in the recipe panel.

**Exported comments:**

When comments are exported in scripts from the Recipe panel, they are in the following format:

```
// This is a comment.
```

## Syntax and Parameters

```
comment comment: 'This is a comment.'
```

| Token | Required? | Data Type | Description |
|-------|-----------|-----------|-------------|
| comment | Y | transform | Name of the transform |
| comment | Y | String | Text of the comment to include. |

**comment**

A text value that represents the comment that will appear in the recipe panel and in all exported scripts.

| Required? | Data Type |
|-----------|-----------|
|  |  |

| | |
|-----|--------|
| Yes | String |

## Examples

> **Tip:** For additional examples, see *Common Tasks.*

# Countpattern Transform

**Contents:**

> **NOTE:** Transforms are a part of the underlying language, which is not directly accessible to users. This content is maintained for reference purposes only. For more information on the user-accessible equivalent to transforms, see *Transformation Reference*.

Counts the number of instances of a specified pattern in a column and writes that value into a newly generated column. Source column is unchanged.

## Basic Usage

```
countpattern col: myCol on: 'honda'
```

**Output:** Generates a new column containing the number of instances of the string `honda` that appear in each row of the column, `myCol`.

## Syntax and Parameters

```
countpattern col:column_ref [ignoreCase:true|false] [after:start_point | from:
start_point] [before:end_point | to:end_point] [on:'exact_match']
```

| Token | Required? | Data Type | Description |
|---|---|---|---|
| countpattern | Y | transform | Name of the transform |
| col | Y | string | Source column name |
| ignoreCase | N | boolean | If `true`, matching is case-insensitive. |

Matching parameters:

> **NOTE:** At least one of the following parameters must be included to specify the pattern to count: `after, before, from, on, to`.

| Token | Required? | Data Type | Description |
|---|---|---|---|
| after | N | string | String literal or pattern that precedes the pattern to match |
| before | N | string | String literal or pattern that appears after the pattern to match |

| from | N | string | String literal or pattern that identifies the start of the pattern to match |
| on | N | string | String literal or pattern that identifies the pattern to match. |
| to | N | string | String literal or pattern that identifies the end of the pattern to match |

For more information on syntax standards, see *Language Documentation Syntax Notes*.

**col**

Identifies the column to which to apply the transform. You can specify only one column.

```
countpattern col: MyCol on: 'MyString'
```

**Output:** Counts the number of instances of the value `MyString` in the `MyCol` column and writes this value to a new column.

**Usage Notes:**

| Required? | Data Type |
| --- | --- |
| Yes | String (column name) |

**after**

```
countpattern col: MyCol after: 'entry:'
```

**Output:** Counts `1` if there is anything that appears in the `MyCol` column value after the string `entry:`. If the value `entry:` does not appear in the column, the output value is `0`.

A pattern identifier that precedes the value or pattern to match. Define the `after` parameter value using string literals, regular expressions, or Patterns .

**Usage Notes:**

| Required? | Data Type |
| --- | --- |
| No | String (string literal or pattern) |

- The `after` and `from` parameters are very similar. `from` includes the matching value as part of the extracted string.
- `after` can be used with either `to`, `on`, or `before`. See *Pattern Clause Position Matching*

**before**

A pattern identifier that occurs after the value or pattern to match. Define the pattern using string literals, regular expressions, or Patterns .

```
countpattern col: MyCol before: '|'
```

**Output:**

- Counts `1` if there is a value that appears before the pipe character (`|`) in the `MyCol` column, and no other pattern parameter is specified. If the `before` value does not appear in the column, the output value is `0`.
- If another pattern parameter such as `after` is specified, the total count of instances is written to the new column.

**Usage Notes:**

| Required? | Data Type |
|---|---|
| No | String or pattern |

- The `before` and `to` parameters are very similar. `to` includes the matching value as part of the extracted string.
- `before` can be used with either `from`, `on`, or `after`. See *Pattern Clause Position Matching* .

### from

Identifies the pattern that marks the beginning of the value to match. It can be a string literal, Pattern , or regular expression. The `from` value is included in the match.

```
countpattern col: MyCol from: 'go:'
```

### Output:

- Counts `1` if contents from `MyCol` that occur from `go:`, to the end of the cell when no other pattern parameter is specified. If `go:` does not appear in the column, the output value is blank.
- If another pattern parameter such as `to` is specified, the total count of instances is written to the new column.

### Usage Notes:

| Required? | Data Type |
|---|---|
| No | String or pattern |

- The `after` and `from` parameters are very similar. `from` includes the matching value as part of the extracted string.
- `from` can be used with either `to` or `before`. See *Pattern Clause Position Matching* .

### on

Identifies the pattern to match, which can be a string literal, Pattern , or regular expression.

```
countpattern col: MyCol on: `###ERROR`
```

> **Tip:** You can insert the Unicode equivalent character for this parameter value using a regular expression of the form `/\uHHHH/`. For example, `/\u0013/` represents Unicode character `0013` (carriage return). For more information, see *Supported Special Regular Expression Characters*.

### Usage Notes:

| Required? | Data Type |
|---|---|
| No | String (literal, regular expression, or Pattern |

### to

Identifies the pattern that marks the ending of the value to match. Pattern can be a string literal,  Patterns , or regular expression. The `to` value is included in the match.

```
countpattern col:MyCol from:'note:' to: `/`
```

**Output:**

- Counts instances from `MyCol` column of all values that begin with `note:` up to a backslash character.
- If a second pattern parameter is not specified, then this value is either `0` or `1`.

**Usage Notes:**

| Required? | Data Type |
|-----------|-----------|
| No | String or pattern |

- The `before` and `to` parameters are very similar. `to` includes the matching value as part of the extracted string.
- `to` can be used with either `from` or `after`. See *Pattern Clause Position Matching*.

### ignoreCase

Indicates whether the match should ignore case or not.

- Set to `true` to ignore case matching.
- (Default) Set to `false` to perform case-sensitive matching.

```
countpattern col: MyCol on: 'My String' ignoreCase: true
```

**Output:** Counts the instances of the following values if they appear in the `MyCol` column: `My String`, `my string`, `My string`, etc.

**Usage Notes:**

| Required? | Data Type |
|-----------|-----------|
| No | Boolean |

## Examples

> **Tip:** For additional examples, see *Common Tasks*.

### Example - counting patterns in tweets

This example demonstrates how to count the number of occurrences of text patterns in a column.

**Functions:**

| Item | Description |
|------|-------------|
| IF Function | The `IF` function allows you to build if/then/else conditional logic within your transforms. |

**Source:**

The dataset below contains fictitious tweet information shortly after the release of an application called, "Myco ExampleApp".

| Date | twitterId | isEmployee | tweet |
|------|-----------|------------|-------|
| 11/5/15 | lawrencetlu38141 | FALSE | Just downloaded Myco ExampleApp! Transforming data in 5 mins! |
| 11/5/15 | petramktng024 | TRUE | Try Myco ExampleApp, our new free data wrangling app! See www.example.com. |
| 11/5/15 | joetri221 | TRUE | Proud to announce the release of Myco ExampleApp, the free version of our enterprise product. Check it out at www.example.com. |
| 11/5/15 | datadaemon994 | FALSE | Great start with Myco ExampleApp. Super easy to use, and actually fun. |
| 11/5/15 | 99redballoons99 | FALSE | Liking this new ExampleApp! Good job, guys! |
| 11/5/15 | bigdatadan7182 | FALSE | @support, how can I find example datasets for use with your product? |

There are two areas of analysis:

- For non-employees, you want to know if they are mentioning the new product by name.
- For employees, you want to know if they are including cross-references to the web site as part of their tweet.

**Transformation:**

The following counts the occurrences of the string `ExampleApp` in the `tweet` column. Note the use of the `ignoreCase` parameter to capture capitalization differences:

| | |
|---|---|
| **Transformation Name** | `Count matches` |
| **Parameter: Column** | `tweet` |
| **Parameter: Option** | `Text or pattern` |
| **Parameter: Text or pattern to count** | `'ExampleApp'` |
| **Parameter: Ignore case** | `true` |

For non-employees, you want to track if they have mentioned the product in their tweet:

| | |
|---|---|
| **Transformation Name** | `New formula` |
| **Parameter: Formula type** | `Single row formula` |
| **Parameter: Formula** | `if(isEmployee=='FALSE' && countpattern_tweet=='1',true, false)` |
| **Parameter: New column name** | `'nonEmployeeExampleAppMentions'` |

The following counts the occurrences of `example.com` in their tweets:

| | |
|---|---|
| **Transformation Name** | `Count matches` |
| **Parameter: Column** | `tweet` |
| **Parameter: Option** | `Text or pattern` |
| **Parameter: Text or pattern to count** | `'example.com'` |

| Parameter: Ignore case | `true` |
|---|---|

For employees, you want to track if they included the above cross-reference in their tweets:

| Transformation Name | `New formula` |
|---|---|
| Parameter: Formula type | `Single row formula` |
| Parameter: Formula | `if(isEmployee=='TRUE' && countpattern_tweet1 == 1, true, false)` |
| Parameter: New column name | `'employeeWebsiteCrossRefs'` |

### Results:

After you delete the two columns tabulating the counts, you end up with the following:

| Date | twitterId | isEmployee | tweet | employeeWebsiteCrossRefs | nonEmployeeExampleAppMentions |
|---|---|---|---|---|---|
| 11/5 /15 | lawrencetl u38141 | FALSE | Just downloaded Myco ExampleApp! Transforming data in 5 mins! | false | true |
| 11/5 /15 | petramktn g024 | TRUE | Try Myco ExampleApp, our new free data wrangling app! See www. example.com. | true | false |
| 11/5 /15 | joetri221 | TRUE | Proud to announce the release of Myco ExampleApp, the free version of our enterprise product. Check it out at www. example.com. | true | false |
| 11/5 /15 | datadaem on994 | FALSE | Great start with Myco ExampleApp. Super easy to use, and actually fun. | false | true |
| 11/5 /15 | 99redballo ons99 | FALSE | Liking this new ExampleApp! Good job, guys! | false | true |
| 11/5 /15 | bigdatada n7182 | FALSE | @support, how can I find example datasets for use with your product? | false | false |

# Deduplicate Transform

> **NOTE:** Transforms are a part of the underlying language, which is not directly accessible to users. This content is maintained for reference purposes only. For more information on the user-accessible equivalent to transforms, see *Transformation Reference*.

Removes exact duplicate rows from your dataset. Duplicate rows are identified by exact, case-sensitive matches between values.

For example, two strings with different capitalization do not match.

## Basic Usage

```
deduplicate
```

**Output:** Rows that are exact duplicates of previous rows are removed from the dataset.

## Syntax and Parameters

There are no parameters for this transform.

## Examples

> **Tip:** For additional examples, see *Common Tasks*.

**Matches and non-matches for Deduplicate Transform**

**Source:**

For example, your dataset looks like the following, which contains three sets of very similar records. The second row of each set is different in one column than the previous one.

| Name | Date | Score |
|------|------|-------|
| Joe Jones | 1/2/03 | 88 |
| joe jones | 1/2/03 | 88 |
| Jane Jackson | 2/3/04 | 77 |
| Jane Jackson | February 3, 2004 | 77 |
| Jill Johns | 3/4/05 | 66 |
| Jill Johns | 3/4/05 | 66.00 |

**Transformation:**

| Transformation Name | Remove duplicate rows |
|---------------------|-----------------------|

If you remove duplicate rows on this dataset, no rows are previewed. This preview indicates that no rows will be removed as duplicates. You might need to clean up the data before you can remove any duplicate rows.

Your first step should be get your capitalization consistent. Try the following:

| | |
|---|---|
| **Transformation Name** | `Edit column with formula` |
| **Parameter: Columns** | `Name` |
| **Parameter: Formula** | `proper(Name)` |

All entries in the `Name` column now appear as proper names. Next, you can clean up the score column by normalizing numeric values to the same format. Try the following:

| | |
|---|---|
| **Transformation Name** | `Edit column with formula` |
| **Parameter: Columns** | `Score` |
| **Parameter: Formula** | `numformat(Score, '##.00')` |

The above transformation normalizes the numeric formats to include two-digits after the decimal point always, which forces all numbers to be the same format. You can use the `##` format string here, too.

Use the following to fix the Date column:

| | |
|---|---|
| **Transformation Name** | `Replace text or pattern` |
| **Parameter: Column** | `Date` |
| **Parameter: Find** | `'February 3, 2004'` |
| **Parameter: Replace with** | `'2/3/04'` |

Now, you can deduplicate your dataset:

| | |
|---|---|
| **Transformation Name** | `Remove duplicate rows` |

**Results:**

| Name | Date | Score |
|---|---|---|
| Joe Jones | 1/2/03 | 88.00 |
| Jane Jackson | 2/3/04 | 77.00 |
| Jill Johns | 3/4/05 | 66.00 |

# Delete Transform

> **NOTE:** Transforms are a part of the underlying language, which is not directly accessible to users. This content is maintained for reference purposes only. For more information on the user-accessible equivalent to transforms, see *Transformation Reference*.

Deletes a set of rows in your dataset, based on a condition specified in the `row` expression. If the conditional expression is `true`, then the row is deleted.

The `delete` transform is the opposite of the `keep` transform. See *Keep Transform*.

## Basic Usage

```
delete row:(dateAge >= 90)
```

**Output:** For each row in the dataset, if the value in the `dateAge` column is greater than or equal to `90`, the row is deleted.

## Syntax and Parameters

```
delete row:(expression)
```

| Token | Required? | Data Type | Description |
|-------|-----------|-----------|-------------|
| delete | Y | transform | Name of the transform |
| row | Y | string | Expression identifying the row or rows to delete. If expression evaluates to `true` for a row, the row is removed. |

For more information on syntax standards, see *Language Documentation Syntax Notes*.

**row**

Expression to identify the row or rows on which to perform the transform. Expression must evaluate to `true` or `false`.

**Examples:**

| Expression | Description |
|------------|-------------|
| `Score >= 50` | `true` if the value in the `Score` column is greater than 50. |
| `LEN(LastName) > 8` | `true` if the length of the value in the `LastName` column is greater than 8. |
| `ISMISSING([Title])` | `true` if the row value in the `Title` column is missing. |
| `ISMISMATCHED(Score,['Integer'])` | `true` if the row value in the `Score` column is mismatched against the Integer data type. |

**Example:**

```
delete row: (lastContactDate < 01/01/2010 || status == 'Inactive')
```

**Output:** Deletes any row in the dataset where the `lastContactDate` is before January 1, 2010 or the status is `Inactive`.

**Usage Notes:**

| Required? | Data Type |
|---|---|
| Yes | Expression that evaluates to `true` or `false` |

## Examples

> **Tip:** For additional examples, see *Common Tasks*.

**Example - Remove old products and keep new orders**

This examples illustrates how you can keep and delete rows from your dataset.

**Source:**

Your dataset includes the following order information. You want to edit your dataset so that:

- All orders for products that are no longer available are removed. These include the following product IDs: `P 100`, `P101`, `P102`, `P103`.
- All orders that were placed within the last 90 days are retained.

| OrderId | OrderDate | ProdId | ProductName | ProductColor | Qty | OrderValue |
|---|---|---|---|---|---|---|
| 1001 | 6/14/2015 | P100 | Hat | Brown | 1 | 90 |
| 1002 | 1/15/2016 | P101 | Hat | Black | 2 | 180 |
| 1003 | 11/11/2015 | P103 | Sweater | Black | 3 | 255 |
| 1004 | 8/6/2015 | P105 | Cardigan | Red | 4 | 320 |
| 1005 | 7/29/2015 | P103 | Sweeter | Black | 5 | 375 |
| 1006 | 12/1/2015 | P102 | Pants | White | 6 | 420 |
| 1007 | 12/28/2015 | P107 | T-shirt | White | 7 | 390 |
| 1008 | 1/15/2016 | P105 | Cardigan | Red | 8 | 420 |
| 1009 | 1/31/2016 | P108 | Coat | Navy | 9 | 495 |

**Transformation:**

First, you remove the orders for old products. Since the set of products is relatively small, you can start first by adding the following:

> **NOTE:** Just preview this transformation. Do not add it to your recipe yet.

| Transformation Name | `Filter rows` |
|---|---|
| **Parameter: Condition** | `Custom formula` |
| **Parameter: Type of formula** | `Custom single` |
| **Parameter: Condition** | `(ProdId == 'P100')` |
| **Parameter: Action** | `Delete matching rows` |

When this step is previewed, you should notice that the top row in the above table is highlighted for removal. Notice how the transformation relies on the `ProdId` value. If you look at the `ProductName` value, you might notice that there is a misspelling in one of the affected rows, so that column is not a good one for comparison purposes.

You can add the other product IDs to the transformation in the following expansion of the transformation, in which any row that has a matching `ProdId` value is removed:

| Transformation Name | `Filter rows` |
|---|---|
| **Parameter: Condition** | `Custom formula` |
| **Parameter: Type of formula** | `Custom single` |
| **Parameter: Condition** | `(ProdId == 'P100' \|\| ProdId == 'P101' \|\| ProdId == 'P102' \|\|`<br>`ProdId == 'P103')` |
| **Parameter: Action** | `Delete matching rows` |

When the above step is added to your recipe, you should see data that looks like the following:

| OrderId | OrderDate | ProdId | ProductName | ProductColor | Qty | OrderValue |
|---|---|---|---|---|---|---|
| 1004 | 8/6/2015 | P105 | Cardigan | Red | 4 | 320 |
| 1007 | 12/28/2015 | P107 | T-shirt | White | 7 | 390 |
| 1008 | 1/15/2016 | P105 | Cardigan | Red | 8 | 420 |
| 1009 | 1/31/2016 | P108 | Coat | Navy | 9 | 495 |

Now, you can filter out of the dataset orders that are older than 90 days. First, add a column with today's date:

| Transformation Name | `New formula` |
|---|---|
| **Parameter: Formula type** | `Single row formula` |
| **Parameter: Formula** | `'2/25/16'` |
| **Parameter: New column name** | `'today'` |

Keep the rows that are within 90 days of this date using the following:

| Transformation Name | `Filter rows` |
|---|---|
| | |

| Parameter: Condition | Custom formula |
|---|---|
| Parameter: Type of formula | Custom single |
| Parameter: Condition | datedif(OrderDate,today,day) <= 90 |
| Parameter: Action | Keep matching rows |

Don't forget to delete the `today` column, which is no longer needed:

| Transformation Name | Delete columns |
|---|---|
| Parameter: Columns | today |
| Parameter: Action | Delete selected columns |

**Results:**

| OrderId | OrderDate | ProdId | ProductName | ProductColor | Qty | OrderValue |
|---|---|---|---|---|---|---|
| 1007 | 12/28/2015 | P107 | T-shirt | White | 7 | 390 |
| 1008 | 1/15/2016 | P105 | Cardigan | Red | 8 | 420 |
| 1009 | 1/31/2016 | P108 | Coat | Navy | 9 | 495 |

# Derive Transform

**Contents:**

---

> **NOTE:** Transforms are a part of the underlying language, which is not directly accessible to users. This content is maintained for reference purposes only. For more information on the user-accessible equivalent to transforms, see *Transformation Reference*.

Generate a new column where the values are the output of the `value` expression. Expression can be calculated based on values specified in the `group` parameter. Output column can be named as needed.

## Basic Usage

**String literal example:**

```
derive type: single value: 'passed' as:'status'
```

**Output:** Generates a new column called `status`, each row of which contains `passed` for its value.

**Column reference example:**

```
derive type: single value:productName as:'orig_productName'
```

**Output:** Generates a new column called `orig_productName`, which contains all of the values in `productName`, effectively serving as a backup of the source column.

**Function reference example:**

```
derive type: single value:SQRT(POW(a,2) + POW(b,2)) as:'c'
```

**Output:** Generates a new column called `c`, which is the calculation of the Pythagorean theorem for the values stored in `a` and `b`. For more information on this example, see *POW Function*.

**Window function example:**

You can use window functions in your `derive` transforms:

```
derive type: multiple col: avgRolling value: ROLLINGAVERAGE(POS_Sales, 7, 0) group:
saleDate order: saleDate
```

**Output:** Calculate the value in the column of `avgRolling` to be the rolling average of the `POS_Sales` values for the preceding seven days, grouped and ordered by the `saleDate` column. For more information, see *Window Functions*.

## Syntax and Parameters

```
derive type: single|multiple value:(expression) [order: order_col] [group: group_col] [as:'
new_column_name']
```

| Token | Required? | Data Type | Description |
|-------|-----------|-----------|-------------|
| derive | Y | transform | Name of the transform |
| type | Y | string | Type of formula: `single` (single row) or `multiple` (multi-row) |
| value | Y | string | Expression that generates the value to store in the new column |
| order | N | string | Column or column names by which to sort the dataset before the `value` expression is applied |
| group | N | string | If you are using aggregate or window functions, you can specify a `group` expression to identify the subset of records to apply the `value` expression. |
| as | N | string | Name of the newly generated column |

For more information on syntax standards, see *Language Documentation Syntax Notes*.

**type**

Type of formula in the transformation:

| Value | Description |
|-------|-------------|
| `single` | Formula calculations are contained within a single row of values. |
| `multiple` | Formula calculations involve multiple rows of inputs. |

**Usage Notes:**

| Required? | Data Type |
|-----------|-----------|
| Yes | String (`single` or `multiple`) |

**value**

Identifies the expression that is applied by the transform. The `value` parameter can be one of the following types:

- test predicates that evaluate to Boolean values (`value: myAge == '30'` yields a `true` or `false` value), or
- computational expressions ( `value: abs(pow(myCol,3))` ).

The expected type of `value` expression is determined by the transform type. Each type of expression can contain combinations of the following:

- **literal values:** `value: 'Hello, world'`
- **column references:** `value: amountOwed * 10`
- **functions:** `value: left(myString, 4)`

- **combinations:** `value: abs(pow(myCol,3))`

The types of any generated values are re-inferred by the platform.

**Usage Notes:**

| Required? | Data Type |
|-----------|-----------|
| Yes | String (literal, column reference, function call, or combination) |

### order

This parameter specifies the column on which to sort the dataset before applying the specified function. For combination sort keys, you can add multiple comma-separated columns.

> **NOTE:** The order parameter must unambiguously specify an ordering for the data, or the generated results may vary between job executions.

> **NOTE:** If it is present, the dataset is first grouped by the `group` value before it is ordered by the values in the `order` column.

> **NOTE:** The `order` column does not need to be sorted before the transform is executed on it.

> **Tip:** To sort in reverse order, prepend the column name with a dash (`-MyDate`).

**Usage Notes:**

| Required? | Data Type |
|-----------|-----------|
| No | String (column name) |

### group

Identifies the column by which the dataset is grouped for purposes of applying the transform.

> **NOTE:** Transforms that use the `group` parameter can result in non-deterministic re-ordering in the data grid. However, you should apply the `group` parameter, particularly on larger datasets, or your job may run out of memory and fail. To enforce row ordering, you can use the `sort` transform. For more information, see *Sort Transform*.

The `ProdId` column contains three values: `P001`, `P002`, and `P003`, and you add the following transformation:

```
derive type: single value:SUM(Sales) group:ProdId as:'SalesByProd'
```

The above transform generates the `SalesByProd` column, which contains the sum of the `Sales` values, as grouped according to the three product identifiers.

If the value parameter contains aggregate or window functions, you can apply the `group` parameter to specify subsets of records across which the value computation is applied.

> **NOTE:** Transforms that use the `group` parameter can result in non-deterministic re-ordering in the data grid. However, you should apply the `group` parameter, particularly on larger datasets, or your job may run out of memory and fail. To enforce row ordering, you can use the `sort` transform. For more information, see *Sort Transform*.

You can specify one or more columns by which to group using comma-separated column references.

**Usage Notes:**

| Required? | Data Type |
|-----------|-----------|
| No | String (column name) |

**as**

Name of the new column that is being generated. If the `as` parameter is not specified, a default name is used.

```
derive type: single value:(colX * colY) as:'areaXY'
```

**Output:** Generates a new column containing the product of the values in columns `colX` and `colY`. New column is explicitly named, `areaXY`.

**Usage Notes:**

| Required? | Data Type |
|-----------|-----------|
| No | String (column name) |

## Examples

> **Tip:** For additional examples, see *Common Tasks*.

### Example - Basic Derive Examples

The following dataset is used for performing some simple statistical analysis using the `derive` transform.

**Source:**

| StudentId | TestNumber | TestScore |
|-----------|------------|-----------|
| S001 | 1 | 78 |
| S001 | 2 | 85 |
| S001 | 3 | 81 |
| S002 | 1 | 84 |
| S002 | 2 | 92 |
| S002 | 3 | 77 |
| S003 | 1 | 83 |
| S003 | 2 | 88 |
| S003 | 3 | 85 |

**Transformation:**

First, you can calculate the total average score across all tests:

| | |
|---|---|
| **Transformation Name** | `New formula` |
| **Parameter: Formula type** | `Single row formula` |
| **Parameter: Formula** | `average(Score)` |
| **Parameter: New column name** | `'avgScore'` |

In their unformatted form, the output values are lengthy. You can edit the above transform to nest the `value` statement with proper formatting using the `numformat` function:

| | |
|---|---|
| **Transformation Name** | `New formula` |
| **Parameter: Formula type** | `Single row formula` |
| **Parameter: Formula** | `numformat(average(Score),'##.00')` |
| **Parameter: New column name** | `'avgScore'` |

You might also be interested to know how individual students fared and to identify which tests caused the greatest challenges for the students:

| | |
|---|---|
| **Transformation Name** | `New formula` |
| **Parameter: Formula type** | `Single row formula` |
| **Parameter: Formula** | `numformat(average(Score),'##.00')` |
| **Parameter: Group rows by** | `StudentId` |
| **Parameter: New column name** | `'avgScorebyStudentId'` |

| | |
|---|---|
| **Transformation Name** | `New formula` |
| **Parameter: Formula type** | `Single row formula` |
| **Parameter: Formula** | `numformat(average(Score),'##.00')` |
| **Parameter: Group rows by** | `TestNumber` |
| **Parameter: New column name** | `'avgScoreByTest'` |

To calculate total scores for each student, add the following. Since each individual test score is a whole number, no rounding formatting is required.

| | |
|---|---|
| **Transformation Name** | `New formula` |
| **Parameter: Formula type** | `Single row formula` |
| **Parameter: Formula** | `sum(Score)` |
| **Parameter: Group rows by** | `StudentId` |

| | |
|---|---|
| **Parameter: New column name** | `'totalScoreByStudentId'` |

**Results:**

| StudentId | TestNumber | TestScore | avgScore | avgScorebyStudentId | ScoreByTest | totalScoreByStudentId |
|---|---|---|---|---|---|---|
| S001 | 1 | 78 | 83.67 | 81.33 | 81.67 | 244 |
| S001 | 2 | 85 | 83.67 | 81.33 | 88.33 | 244 |
| S001 | 3 | 81 | 83.67 | 81.33 | 81.00 | 244 |
| S002 | 1 | 84 | 83.67 | 84.33 | 81.67 | 253 |
| S002 | 2 | 92 | 83.67 | 84.33 | 88.33 | 253 |
| S002 | 3 | 77 | 83.67 | 84.33 | 81.00 | 253 |
| S003 | 1 | 83 | 83.67 | 85.33 | 81.67 | 256 |
| S003 | 2 | 88 | 83.67 | 85.33 | 88.33 | 256 |
| S003 | 3 | 85 | 83.67 | 85.33 | 81.00 | 256 |

### Example - Rounding Functions

This example demonstrates how the rounding functions work together.

**Functions**:

| Item | Description |
|---|---|
| FLOOR Function | Computes the largest integer that is not more than the input value. Input can be an Integer, a Decimal, a column reference, or an expression. |
| CEILING Function | Computes the **ceiling** of a value, which is the smallest integer that is greater than the input value. Input can be an Integer, a Decimal, a column reference, or an expression. |
| ROUND Function | Rounds input value to the nearest integer. Input can be an Integer, a Decimal, a column reference, or an expression. Optional second argument can be used to specify the number of digits to which to round. |
| MOD Function | Returns the modulo value, which is the remainder of dividing the first argument by the second argument. Equivalent to the % operator. |

**Source:**

| rowNum | X |
|---|---|
| 1 | -2.5 |
| 2 | -1.2 |
| 3 | 0 |
| 4 | 1 |
| 5 | 1.5 |
| 6 | 2.5 |
| 7 | 3.9 |
| 8 | 4 |
| 9 | 4.1 |
| 10 | 11 |

**Transformation:**

| Transformation Name | New formula |
|---|---|
| **Parameter: Formula type** | Single row formula |
| **Parameter: Formula** | FLOOR(X) |
| **Parameter: New column name** | 'floorX' |

| Transformation Name | New formula |
|---|---|
| **Parameter: Formula type** | Single row formula |
| **Parameter: Formula** | CEILING(X) |
| **Parameter: New column name** | 'ceilingX' |

| Transformation Name | New formula |
|---|---|
| **Parameter: Formula type** | Single row formula |
| **Parameter: Formula** | ROUND (X) |
| **Parameter: New column name** | 'roundX' |

| Transformation Name | New formula |
|---|---|
| **Parameter: Formula type** | Single row formula |
| **Parameter: Formula** | (X % 2) |
| **Parameter: New column name** | 'modX' |

**Results:**

| rowNum | X | modX | roundX | ceilingX | floorX |
|---|---|---|---|---|---|
| 1 | -2.5 | | -2 | -2 | -3 |
| 2 | -1.2 | | -1 | -1 | -2 |
| 3 | 0 | 0 | 0 | 0 | 0 |
| 4 | 1 | 1 | 1 | 1 | 1 |
| 5 | 1.5 | | 2 | 2 | 1 |
| 6 | 2.5 | | 3 | 3 | 2 |
| 7 | 3.9 | | 4 | 4 | 3 |
| 8 | 4 | 0 | 4 | 4 | 4 |
| 9 | 4.1 | | 4 | 5 | 4 |
| 10 | 11 | 1 | 11 | 11 | 11 |

# Drop Transform

> **NOTE:** Transforms are a part of the underlying language, which is not directly accessible to users. This content is maintained for reference purposes only. For more information on the user-accessible equivalent to transforms, see *Transformation Reference*.

Removes the specified column or columns permanently from your dataset.

**Tips:**

- This transform might be automatically applied as one of the first steps of your recipe.
- If you want to hide a column from view, select **Hide** from the column drop-down. Note that the data can still be referenced in your transforms and appears in any generated output.
- If you are working with large datasets, you might want to delete columns at the beginning of your recipe, which can assist application and job execution performance. Use the tilde operator to specify ranges of columns.
- You can also specify the columns that you wish to retain and then add the `Keep` action to delete all other columns in the dataset.

## Basic Usage

**Single-column example:**

```
drop col:ThisOldColumn action: Drop
```

**Output:** Deletes the column named `ThisOldColumn`.

**Multi-column example:**

You can specify comma-separated sets of columns.

```
drop col: FirstName, MiddleInitial action: Drop
```

**Output:** Deletes the columns `FirstName` and `MiddleInitial` from your dataset.

**Keep example:**

The following transform keeps the listed columns and deletes all others in the dataset:

```
drop col: FirstName, MiddleInitial action: Keep
```

**Output:** Dataset only contains `FirstName` and `MiddleInitial` columns.

**Column range example:**

You can also specify ranges of columns using the tilde (~) operator:

```
drop col:Column1~Column20 action: Drop
```

**Output:** Deletes the columns `Column1` and `Column20` and all columns displayed in between them in the data grid.

## Syntax and Parameters

```
drop col:column_ref action: [Drop|Keep]
```

| Token | Required? | Data Type | Description |
| --- | --- | --- | --- |
| drop | Y | transform | Name of the transform |
| col | Y | string | Name of the column or expression for columns to delete |
| action | Y | string | `Drop` or `Keep` the listed columns |

For more information on syntax standards, see *Language Documentation Syntax Notes*.

**col**

Identifies the column or columns to which to apply the transform. You can specify one column or more columns.

To specify multiple columns:

- Discrete column names are comma-separated.
- Values for column names are case-sensitive.

**Usage Notes:**

| Required? | Data Type |
| --- | --- |
| Yes | String (column name) |

**action**

Identifies whether the action performed by the transformation:

- `Drop` - Listed columns are deleted from the dataset.
- `Keep` - Listed columns are retained in the dataset, and all other columns are deleted.

**Usage Notes:**

| Required? | Data Type |
| --- | --- |
| Yes | String (`Drop` or `Keep`) |

Examples

> **Tip:** For additional examples, see *Common Tasks*.

See above.

# Extract Transform

**Contents:**

---

> **NOTE:** Transforms are a part of the underlying language, which is not directly accessible to users. This content is maintained for reference purposes only. For more information on the user-accessible equivalent to transforms, see *Transformation Reference*.

Extracts a subset of data from one column and inserts it into a new column, based on a specified string or pattern. The source column in unmodified.

> **Tip:** Use the `extract` transform if you need to retain the source column. Otherwise, you might be able to use the `split` transform. See *Split Transform*.

## Basic Usage

```
extract col: text on: 'honda' limit: 10
```

**Output:** Extracts the value `honda` from the source column `text` up to 10 times and insert into a new column. The source column `text` is unmodified.

## Syntax and Parameters

```
extract col:column_ref [quote:'quoted_string'] [ignoreCase:true|false] [limit:max_count]
[after:start_point | from: start_point] [before:end_point | to:end_point]
[on:'exact_match'] [at:(start_index,end_index)]
```

> **NOTE:** At least one of the following parameters must be included to specify the pattern to extract: `after`, `at`, `before`, `from`, `on`, `to`.

| Token | Required? | Data Type | Description |
|---|---|---|---|
| extract | Y | transform | Name of the transform |

| | | | | |
|---|---|---|---|---|
| col | Y | string | Source column name | |
| quote | N | string | Specifies a quoted object that is omitted from pattern matching | |
| ignoreCase | N | boolean | If `true`, matching is case-insensitive. | |
| limit | N | integer (positive) | Identifies the number of extractions that can be performed from a single value. Default is 1. | |

Matching parameters:

| Parameter | Required? | Data Type | Description |
|---|---|---|---|
| after | N | string | String literal or pattern that precedes the pattern to match |
| at | N | Array | Two-integer array identifying the character indexes of start and end characters to match |
| before | N | string | String literal or pattern that appears after the pattern to match |
| from | N | string | String literal or pattern that identifies the start of the pattern to match |
| on | N | string | String literal or pattern that identifies the pattern to match. |
| to | N | string | String literal or pattern that identifies the end of the pattern to match |

For more information on syntax standards, see *Language Documentation Syntax Notes*.

**col**

Identifies the column to which to apply the transform. You can specify only one column.

```
extract col: MyCol on: 'MyString'
```

**Output:** Extracts value `My String` in new column if it is present in `MyCol`. Otherwise, new column value is blank.

**Usage Notes:**

| Required? | Data Type |
|---|---|
| Yes | String (column name) |

**after**

```
extract col: MyCol after: 'Important:'
```

**Output:** Extracts value in `MyCol` that appears after the string `Important:`. If the `after` value does not appear in the column, the output value is blank.

A pattern identifier that precedes the value or pattern to match. Define the `after` parameter value using string literals, regular expressions, or Patterns .

**Usage Notes:**

| Required? | Data Type |
|---|---|
| No | String or pattern |

- The `after` and `from` parameters are very similar. `from` includes the matching value as part of the extracted string.
- `after` can be used with either `to`, `on`, or `before`. See *Pattern Clause Position Matching*.

**at**

```
extract col: MyCol at: 2,6
```

**Output:** Extracts contents of `MyCol` that starts at the second character in the column and extends to the sixth character of the column.

Identifies the start and end point of the pattern to interest.

Parameter inputs are in the form of `x,y` where `x` and `y` are positive integers indicating the starting character and ending character, respectively, of the pattern of interest.

- `x` must be less than `y`.
- If `y` is greater than the length of the value, the pattern is defined to the end of the value, and a match is made.

**Usage Notes:**

| Required? | Data Type |
|-----------|-----------|
| No | Array of two Integers ( X , Y ) |

The `at` parameter cannot be combined with any of the following: `on`, `after`, `before`, `from`, `to`, and `quote`. See *Pattern Clause Position Matching*.

**before**

```
extract col: MyCol before: '|'
```

**Output:** Extracts contents of `MyCol` that occur before the pipe character (|). If the `before` value does not appear in the column, the output value is blank.

A pattern identifier that occurs after the value or pattern to match. Define the pattern using string literals, regular expressions, or Patterns .

**Usage Notes:**

| Required? | Data Type |
|-----------|-----------|
| No | String or pattern |

- The `before` and `to` parameters are very similar. `to` includes the matching value as part of the extracted string.
- `before` can be used with either `from`, `on`, or `after`. See *Pattern Clause Position Matching* .

**from**

```
extract col: MyCol from: 'go:'
```

**Output:** Extracts contents from `MyCol` that occur after `go:`, including `go:`. If the `from` value does not appear in the column, the output value is blank.

Identifies the pattern that marks the beginning of the value to match. It can be a string literal, Pattern , or regular expression. The `from` value is included in the match.

**Usage Notes:**

| Required? | Data Type |
|-----------|-----------|
| No | String or pattern |

- The `after` and `from` parameters are very similar. `from` includes the matching value as part of the extracted string.
- `from` can be used with either `to` or `before`. See *Pattern Clause Position Matching* .

**on**

```
extract col: MyCol on: `###ERROR`
```

Identifies the pattern to match, which can be a string literal, Pattern , or regular expression.

> **Tip:** You can insert the Unicode equivalent character for this parameter value using a regular expression of the form `/\uHHHH/`. For example, `/\u0013/` represents Unicode character `0013` (carriage return). For more information, see *Supported Special Regular Expression Characters*.

**Usage Notes:**

| Required? | Data Type |
|-----------|-----------|
| No | String (literal, regular expression, or Trifacta pattern ) |

**to**

```
extract col:MyCol from:'note:' to: `{end}`
```

**Output:** Extracts from `MyCol` column all values that begin with `note:` up to the end of the value.

Identifies the pattern that marks the ending of the value to match. Pattern can be a string literal, Patterns , or regular expression. The `to` value is included in the match.

**Usage Notes:**

| Required? | Data Type |
|-----------|-----------|
| No | String or pattern |

- The `before` and `to` parameters are very similar. `to` includes the matching value as part of the extracted string.
- `to` can be used with either `from` or `after`. See *Pattern Clause Position Matching*.

**quote**

```
extract col: MyCol quote: 'First' after: `{start}%?`
```

**Output:** Extracts each cell value from the `MyCol` column, starting at the second character in the cell, as long as the string `First` does not appear in the cell.

Can be used to specify a string as a single quoted object. This parameter value can be one or more characters.

**Usage Notes:**

| Required? | Data Type |
|-----------|-----------|
| | |

| No | String |
|----|--------|

- Parameter value is the quoted object.
- The `quote` value can appear anywhere in the column value. It is not limited by the constraints of any other parameters.

**ignoreCase**

```
extract col: MyCol on: 'My String' ignoreCase: true
```

**Output:** Extracts the following values if they appear in the `MyCol` column: `My String`, `my string`, `My string`, etc.

Indicates whether the match should ignore case or not.

- Set to `true` to ignore case matching.
- (Default) Set to `false` to perform case-sensitive matching.

**Usage Notes:**

| Required? | Data Type |
|-----------|-----------|
| No | Boolean |

**limit**

```
extract col: MyCol on: 'z' limit: 3
```

**Output:** Extracts each instance of the letter `z` in the `MyCol` column into a separate column, generating up to 3 new columns.

The `limit` parameter defines the maximum number of times that a pattern can be matched within a column.

> **NOTE:** The `limit` parameter cannot be used with the following parameters: `at`, `positions`, or `delimiters`.

A set of new columns is generated, as defined by the `limit` parameter. Each matched instance populates a separate column, until there are no more matches or all of the `limit`-generated new columns are filled.

**Usage Notes:**

| Required? | Data Type |
|-----------|-----------|
| No | Integer (positive) |

- Defines the maximum number of columns that can be created by the `extract` transform.
- If not specified, exactly one column is created.

## Examples

> **Tip:** For additional examples, see *Common Tasks*.

## Example - Extract First Name

**Source:**

| Name |
| --- |
| Mr. Mike Smith |
| Dr Jane Jones |
| Miss Meg Moon |

**Transformation:**

The following transformation extracts the second word in the above dataset. Content is extracted after the first space and before the next space.

> **Tip:** If you want to break out salutation, first name, and last name at the same time, you should use the Split Column transformation instead.

| Transformation Name | `Extract text or pattern` |
| --- | --- |
| **Parameter: Column to extract from** | `Name` |
| **Parameter: Option** | `Custom text or pattern` |
| **Parameter: Start extracting after** | `' '` |
| **Parameter: End extracting before** | `' '` |

**Results:**

| Name | Name2 |
| --- | --- |
| Mr. Mike Smith | Mike |
| Dr Jane Jones | Jane |
| Miss Meg Moon | Meg |

## Example - Extract Log Levels

**Source:**

The following represents raw log messages extracted from an application. You want to extract the error level for each message: `INFO`, `WARNING`, or `ERROR`.

| app_log |
| --- |
| 20115-10-30T15:43:37:874Z INFO Client env:started |
| 20115-10-30T15:43:38:009Z INFO Client env:launched Chromium component |
| 20115-10-30T15:43:38:512Z ERROR Client env:failed to connect to local DB |
| 20115-10-30T15:43:38:515Z INFO Client env:launched application |

**Transformation:**

The text of interest appears after the timestamp and before the message.

- In the `after` clause, a pattern is required. In this case, the selection rule identifies the last segment of the timestamp, with the three pound signs (#) identifying three digits of unknown value. The "Z " value gives the selection rule an extra bit of specificity. Note the backticks to denote the selection rule.
- In the `before` clause, you can use a simple space character string, since it is consistent across all of the data.

| Transformation Name | Extract text or pattern |
|---|---|
| Parameter: Column to extract from | app_log |
| Parameter: Option | Custom text or pattern |
| Parameter: Start extracting after | `` `###Z ` `` |
| Parameter: End extracting before | `' '` |

**Results:**

| app_log | app_log_2 |
|---|---|
| 20115-10-30T15:43:37:874Z INFO Client env:started | INFO |
| 20115-10-30T15:43:38:009Z INFO Client env:launched Chromium component | INFO |
| 20115-10-30T15:43:38:512Z ERROR Client env:failed to connect to local DB | ERROR |
| 20115-10-30T15:43:38:515Z INFO Client env:launched application | INFO |

### Example - Clean up marketing contact data with replace, set, and extract

This example illustrates the different uses of the replacement transformations to replace or extract cell data.

**Source:**

The following dataset contains contact information that has been gathered by your marketing platform from actions taken by visitors on your website. You must clean up this data and prepare it for use in an analytics platform.

| LeadId | LastName | FirstName | Title | Phone | Request |
|---|---|---|---|---|---|
| LE160301001 | Jones | Charles | Chief Technical Officer | 415-555-1212 | reg |
| LE160301002 | Lyons | Edward | | 415-012-3456 | download whitepaper |
| LE160301003 | Martin | Mary | CEO | 510-555-5555 | delete account |
| LE160301004 | Smith | Talia | Engineer | 510-123-4567 | free trial |

**Transformation:**

**Title column:** For example, you first notice that some data is missing. Your analytics platform recognizes the string value, "`#MISSING#`" as an indicator of a missing value. So, you click the missing values bar in the Title column. Then, you select the Replace suggestion card. Note that the default replacement is a null value, so you click **Edit** and update it:

| Transformation Name | Edit column with formula |
|---|---|
| Parameter: Columns | Title |
| Parameter: Formula | if(ismissing([Title]),'#MISSING#',Title) |

**Request column:** In the Request column, you notice that the `reg` entry should be cleaned up. Add the following transformation, which replaces that value:

| Transformation Name | `Replace text or pattern` |
|---|---|
| Parameter: Column | `Request` |
| Parameter: Find | `` `{start}reg{end}` `` |
| Parameter: Replace with | `Registration` |

The above transformation uses a Pattern as the expression of the `on:` parameter. This expression indicates to match from the start of the cell value, the string literal `reg`, and then the end of the cell value, which matches on complete cell values of `reg` only.

This transformation works great on the sample, but what happens if the value is `Reg` with a capital `R`? That value might not be replaced. To improve the transformation, you can modify the transformation with the following Pattern in the `on` parameter, which captures differences in capitalization:

| Transformation Name | `Replace text or pattern` |
|---|---|
| Parameter: Column | `Request` |
| Parameter: Find | `` `{start}{[R|r]}eg{end}` `` |
| Parameter: Replace with | `'Registration'` |

Add the above transformation to your recipe. Then, it occurs to you that all of the values in the `Request` column should be capitalized in title or proper case:

| Transformation Name | `Edit column with formula` |
|---|---|
| Parameter: Columns | `Request` |
| Parameter: Formula | `proper(Request)` |

Now, all values are capitalized as titles.

**Phone column:** You might have noticed some issues with the values in the `Phone` column. In the United States, the prefix `555` is only used for gathering information; these are invalid phone numbers.

In the data grid, you select the first instance of `555` in the column. However, it selects all instances of that pattern, including ones that you don't want to modify. In this case, continue your selection by selecting the similar instance of `555` in the other row. In the suggestion cards, you click the Replace Text or Pattern transformation.

Notice, however, that the default Replace Text or Pattern transformation has also highlighted the second `555` pattern in one instance, which could be a problem in other phone numbers not displayed in the sample. You must modify the selection pattern for this transformation. In the `on:` parameter below, the Pattern has been modified to match only the instances of `555` that appear in the second segment in the phone number format:

| Transformation Name | `Replace text or pattern` |
|---|---|
| Parameter: Column | `Phone` |
| Parameter: Find | `` `{start}%{3}-555-%*{end}` `` |
| Parameter: Replace with | `'#INVALID#'` |

| | |
|---|---|
| **Parameter: Match all occurrences** | `true` |

Note the wildcard construct has been added (`%*`). While it might be possible to add a pattern that matches on the last four characters exactly (`%{4}`), that matching pattern would not capture the possibility of a phone number having an extension at the end of it. The above expression does.

> **NOTE:** The above transformation creates values that are mismatched with the Phone Number data type. In this example, however, these mismatches are understood to be for the benefit of the system consuming your Trifacta output.

**LeadId column:** You might have noticed that the lead identifier column (`LeadId`) contains some embedded information: a date value and an identifier for the instance within the day. The following steps can be used to break out this information. The first one creates a separate working column with this information, which allows us to preserve the original, unmodified column:

| | |
|---|---|
| **Transformation Name** | `New formula` |
| **Parameter: Formula type** | `Single row formula` |
| **Parameter: Formula** | `LeadId` |
| **Parameter: New column name** | `'LeadIdworking'` |

You can now work off of this column to create your new ones. First, you can use the following replace transformation to remove the leading two characters, which are not required for the new columns:

| | |
|---|---|
| **Transformation Name** | `Replace text or pattern` |
| **Parameter: Column** | `LeadIdworking` |
| **Parameter: Find** | `'LE'` |
| **Parameter: Replace with** | `''` |

Notice that the date information is now neatly contained in the first characters of the working column. Use the following to extract these values to a new column:

| | |
|---|---|
| **Transformation Name** | `Extract text or pattern` |
| **Parameter: Column to extract from** | `LeadIdworking` |
| **Parameter: Option** | `Custom text or pattern` |
| **Parameter: Text to extract** | `` `{start}%{6}` `` |

The new `LeadIdworking2` column now contains only the date information. Cleaning up this column requires reformatting the data, retyping it as a Datetime type, and then applying the `dateformat` function to format it to your satisfaction. These steps are left as a separate exercise.

For now, let's just rename the column:

| | |
|---|---|
| **Transformation Name** | `Rename columns` |
| **Parameter: Option** | `Manual rename` |
| | |

| Parameter: Column | LeadIdworking1 |
|---|---|
| Parameter: New column name | 'LeadIdDate' |

In the first working column, you can now remove the date information using the following:

| Transformation Name | Replace text or pattern |
|---|---|
| Parameter: Column | LeadIdworking |
| Parameter: Find | `{start}%{6}` |
| Parameter: Replace with | '' |

You can rename this column to indicate it is a daily identifier:

| Transformation Name | Rename columns |
|---|---|
| Parameter: Option | Manual rename |
| Parameter: Column | LeadIdworking |
| Parameter: New column name | 'LeadIdDaily' |

**Results:**

| LeadId | LeadIdDaily | LeadIdDate | LastName | FirstName | Title | Phone | Request |
|---|---|---|---|---|---|---|---|
| LE160301001 | 001 | 160301 | Jones | Charles | Chief Technical Officer | #INVALID# | Registration |
| LE160301002 | 002 | 160301 | Lyons | Edward | #MISSING# | 415-012-3456 | Download Whitepaper |
| LE160301003 | 003 | 160301 | Martin | Mary | CEO | #INVALID# | Delete Account |
| LE160301004 | 004 | 160301 | Smith | Talia | Engineer | 510-123-4567 | Free Trial |

# Extractkv Transform

**Contents:**

- *Basic Usage*
- *Syntax and Parameters*
  - *col*
  - *delimiter*
  - *key*
  - *valueafter*
  - *as*
- *Examples*
  - *Example - extracting key values from car data and the unnesting into separate columns*

---

> **NOTE:** Transforms are a part of the underlying language, which is not directly accessible to users. This content is maintained for reference purposes only. For more information on the user-accessible equivalent to transforms, see *Transformation Reference*.

Extracts key-value pairs from a source column and writes them to a new column.

Source column must be of String type, although the data can be formatted as other data types. The generated column is of Object type.

Your source column (`MyKeyValues`) is formatted in the following manner:

```
key1=value1,key2=value2
```

## Basic Usage

The following transform extracts the key-value pairs. The `key` parameter contains a single pattern that matches all keys that you want to extract:

```
extractkv col: MyKeyValues key:`{alpha}+{digit}` valueafter: '=' delimiter: ','
```

**Output:** The generated column contains data that looks like the following:

```
{"key1":"value1","key2":"value2"}
```

If the source data contained additional keys which were not specified in the transform, those key-value pairs would not appear in the generated column.

## Syntax and Parameters

```
extractkv col:column_ref delimiter:string_literal_pattern key:string_literal_pattern
valueafter:string_literal_pattern [as:'new_column_name']
```

| Parameter | Required? | Data Type | Description |
|-----------|-----------|-----------|-------------|
| extractkv | Y | transform | Name of the transform |
| col | Y | string | Source column name |
| | | | |

| delimiter | Y | string | String literal or pattern that identifies the separator between key-value pairs |
| key | Y | string | Pattern that identifies the key to match |
| valueafter | Y | string | String literal or pattern after which is located a key's value |
| as | N | string | Name of the newly generated column |

For more information on syntax standards, see *Language Documentation Syntax Notes*.

**col**

Identifies the column to which to apply the transform. You can specify only one column.

**Usage Notes:**

| Required? | Data Type |
|-----------|-----------|
| Yes | String (column name) |

**delimiter**

Specifies the character or pattern that defines the end of a key-value pair. This value can be specified as a String literal, regular expression, or  Pattern .

In the following:

```
{ key1=value1,key2=value2 }
```

The delimiter is the comma ( `,` ). The final key-value pair does not need a delimiter.

> **Tip:** You can insert the Unicode equivalent character for this parameter value using a regular expression of the form `/\uHHHH/`. For example, `/\u0013/` represents Unicode character `0013` (carriage return). For more information, see *Supported Special Regular Expression Characters*.

**Usage Notes:**

| Required? | Data Type |
|-----------|-----------|
| Yes | String (literal, regular expression, or Pattern ) |

**key**

Specifies the pattern used to extract the keys from a source column by the `extractkv` transform. For the following data:

```
{ key1=value1,key2=value2 }
```

The keys are represented in the transform by the following parameter and value:

```
key:`{alpha}+{digit}`
```

This pattern matches all keys that begin with a letter and end with a digit. If the source data contains other keys, they do not appear in the extracted data.

**Usage Notes:**

| Required? | Data Type |
|---|---|
| Yes | Single pattern representing the individual keys to extract. |

**valueafter**

Specifies the character or pattern after which the value is specified in a key-value pair. This value can be specified as a String literal, regular expression, or Patterns .

For the following:

```
{ key1=value1,key2=value2 }
```

The `valueafter` string is the equals sign ( `'='`).

**Usage Notes:**

| Required? | Data Type |
|---|---|
| Yes | String (literal, regular expression, or Pattern ) |

**as**

Name of the new column that is being generated. If the `as` parameter is not specified, a default name is used.

**Usage Notes:**

| Required? | Data Type |
|---|---|
| No | String (column name) |

## Examples

> **Tip:** For additional examples, see *Common Tasks*.

**Example - extracting key values from car data and the unnesting into separate columns**

This example shows how you can unpack data nested in an Object into separate columns.

**Source:**

You have the following information on used cars. The `VIN` column contains vehicle identifiers, and the `Properti es` column contains key-value pairs describing characteristics of each vehicle. You want to unpack this data into separate columns.

| VIN | Properties |
|---|---|
| XX3 JT4522 | year=2004,make=Subaru,model=Impreza,color=green,mileage=125422,cost=3199 |
| HT4 UJ9122 | year=2006,make=VW,model=Passat,color=silver,mileage=102941,cost=4599 |
| KC2 WZ9231 | year=2009,make=GMC,model=Yukon,color=black,mileage=68213,cost=12899 |
|  |  |

| LL8 UH4921 | year=2011,make=BMW,model=328i,color=brown,mileage=57212,cost=16999 |

**Transformation:**

Add the following transformation, which identifies all of the key values in the column as beginning with alphabetical characters.

- The `valueafter` string identifies where the corresponding value begins after the key.
- The `delimiter` string indicates the end of each key-value pair.

| Transformation Name | Convert keys/values into Objects |
|---|---|
| **Parameter: Column** | Properties |
| **Parameter: Key** | `` `{alpha}+` `` |
| **Parameter: Separator between key and value** | `` `=` `` |
| **Parameter: Delimiter between pair** | `','` |

Now that the Object of values has been created, you can use the `unnest` transform to unpack this mapped data. In the following, each key is specified, which results in separate columns headed by the named key:

> **NOTE**: Each key must be entered on a separate line in the Path to elements area.

| Transformation Name | Unnest Objects into columns |
|---|---|
| **Parameter: Column** | extractkv_Properties |
| **Parameter: Paths to elements** | year |
| **Parameter: Paths to elements** | make |
| **Parameter: Paths to elements** | model |
| **Parameter: Paths to elements** | color |
| **Parameter: Paths to elements** | mileage |
| **Parameter: Paths to elements** | cost |

**Results:**

When you delete the unnecessary Properties columns, the dataset now looks like the following:

| VIN | year | make | model | color | mileage | cost |
|---|---|---|---|---|---|---|
| XX3 JT4522 | 2004 | Subaru | Impreza | green | 125422 | 3199 |
| HT4 UJ9122 | 2006 | VW | Passat | silver | 102941 | 4599 |
| KC2 WZ9231 | 2009 | GMC | Yukon | black | 68213 | 12899 |
| LL8 UH4921 | 2011 | BMW | 328i | brown | 57212 | 16999 |

# Extractlist Transform

**Contents:**

---

> **NOTE:** Transforms are a part of the underlying language, which is not directly accessible to users. This content is maintained for reference purposes only. For more information on the user-accessible equivalent to transforms, see *Transformation Reference*.

Extracts a set of values based on a specified pattern from a source column of any data type. The generated column contains an array of occurrences of the specified pattern.

While the new column contains array data, the data type of the new column is sometimes inferred as String.

## Basic Usage

Your source column (`myWidgetInventory`) is formatted in the following manner:

```
{ "red":"100","white":"1300","blue":"315","purple":"55"}
```

The following step extracts the raw inventory contents of each color:

```
extractlist col: myWidgetInventory on:`{digit}+`
```

**Output:** The generated column contains data that looks like the following array:

```
["100","1300","315","55"]
```

## Syntax and Parameters

```
extractlist: col:column_ref on:string_literal_pattern delimiter:string_literal_pattern
[quote:'quoted_string'] [as:'new_column_name']
```

| Token | Required? | Data Type | Description |
|-------|-----------|-----------|-------------|
| extractlist | Y | transform | Name of the transform |
| col | Y | string | Source column name |
| on | Y | string | String literal or pattern that identifies the values to extract from the source column |
| | | | |

| | | | |
|---|---|---|---|
| delimiter | Y | string | String literal or pattern that identifies the separator between the values to extract |
| quote | N | string | Specifies a quoted object that is omitted from matching delimiters |
| as | N | string | Name of the newly generated column |

For more information on syntax standards, see *Language Documentation Syntax Notes*.

**col**

Identifies the column to which to apply the transform. You can specify only one column.

**Usage Notes:**

| Required? | Data Type |
|---|---|
| Yes | String (column name) |

**on**

Identifies the pattern to match, which can be a string literal, Pattern , or regular expression.

> **Tip:** You can insert the Unicode equivalent character for this parameter value using a regular expression of the form `/\uHHHH/`. For example, `/\u0013/` represents Unicode character `0013` (carriage return). For more information, see *Supported Special Regular Expression Characters*.

For the `extractlist` tranform, all instances that match this pattern in the source column are extracted into the array list in the new column. Each occurrence in the generated array corresponds to an individual instance in the source; the new column can contain duplicate values.

To create array elements based only on the `delimiter` parameter, set the following regular expression:

```
on:`/+/`
```

**Usage Notes:**

| Required? | Data Type |
|---|---|
| Yes | String (literal, regular expression, or Trifacta® pattern ) |

**delimiter**

Specifies the character or pattern that defines the end of a key-value pair. This value can be specified as a String literal, regular expression, or  Pattern .

In the following:

```
{ key1=value1,key2=value2 }
```

The delimiter is the comma ( `','` ). The final key-value pair does not need a delimiter.

For this transform, this parameter defines the pattern that separates the values that you want to extract into the array.

> **Tip:** You can insert the Unicode equivalent character for this parameter value using a regular expression of the form `/\uHHHH/`. For example, `/\u0013/` represents Unicode character `0013` (carriage return). For more information, see *Supported Special Regular Expression Characters*.

**Usage Notes:**

| Required? | Data Type |
|-----------|-----------|
| Yes | String (literal, regular expression, or Trifacta pattern ) |

**quote**

```
extractlist col: MySourceValues on:`{alpha}+` delimiter:';' quote:'\"'
```

**Output:** Extracts from the `MySourceValues` column each instance of a string value that occurs before the delimiter. Values between double-quotes are considered string literals and are not processed according to the delimiters defined in the transform.

Can be used to specify a string as a single quoted object. This parameter value can be one or more characters.

**Usage Notes:**

| Required? | Data Type |
|-----------|-----------|
| No | String |

- The `quote` value can appear anywhere in the column value. It is not limited by the constraints of any other parameters.

**as**

Name of the new column that is being generated. If the `as` parameter is not specified, a default name is used.

**Usage Notes:**

| Required? | Data Type |
|-----------|-----------|
| No | String (column name) |

## Examples

> **Tip:** For additional examples, see *Common Tasks*.

**Extract hashtag values**

In this example, you extract one or more values from a source column and assemble them in an Array column.

Suppose you need to extract the hashtags from customer tweets to another column. In such cases, you can use the `{hashtag}` Trifacta pattern to extract all hashtag values from a customer's tweets into a new column.

**Source**:

The following dataset contains customer tweets across different locations.

| User Name | Location | Customer tweets |
|---|---|---|
| James | U.K | Excited to announce that we've transitioned Wrangler from a hybrid desktop application to a completely cloud-based service! #dataprep #businessintelligence #CommitToCleanData # London |
| Mark | Berlin | Learnt more about the importance of identifying issues in your data—early and often #CommitToCleanData #predictivetransformations #realbusinessintelligence |
| Catherine | Paris | Clean data is the foundation of your analysis. Learn more about what we consider the five tenets of sound #dataprep, starting with #1a prioritizing and setting targets.  #startwiththeuser #realbusinessintelligence #Paris |
| Dave | New York | Learn how #NewYorklife<br><br>onboarded as part of their #bigdata  #dataprep initiative to unlock hidden insights and make them accessible across departments. |
| Christy | San Francisco | How can you quickly determine the number of times a user ID appears in your data?#dataprep #pivot #aggregation#machinelearning initiatives #SFO |

**Transformation**:

The following transformation extracts the hashtag messages from customer tweets.

| Transformation Name | `Extract matches into Array` |
|---|---|
| **Parameter: Column** | `customer_tweets` |
| **Parameter: Pattern matching elements in the list** | `` `{hashtag}` `` |
| **Parameter: New column name** | `Hashtag tweets` |

Then, the source column can be deleted.

**Results**:

| User Name | Location | Hashtag tweets |
|---|---|---|
| James | U.K | ["#dataprep", "#businessintelligence", "#CommitToCleanData", " # London"] |
| Mark | Berlin | ["#CommitToCleanData",  "#predictivetransformations", "#realbusinessintelligence", "0"] |
| Catherine | Paris | ["#dataprep", "#startwiththeuser","#realbusinessintelligence", "# Paris"] |
| Dave | New York | ["#NewYorklife", "dataprep", "bigdata", "0"] |
| Christy | SanFrancisco | [ "dataprep", "#pivot", "#aggregation", "#machinelearning"] |

**Extract query parameters from URLs**

**Source:**

In this example, a list of URLs identifies the items in the shopping carts of visitors to your web site. You want to extract the shopping cart information embedded in the query parameters of the URL.

| Username | cartURL |
|---|---|
| joe.robinson | http://example123.com/cart.asp?prodid=1001&qty=2 |
| steph.schmidt | http://example123.com/cart.asp?prodid=1005&qty=4 |
| | |

| jack.holmes | http://example123.com/cart.asp?prodid=2102&qty=1 |
|---|---|
| tina.jones | http://example123.com/cart.asp?prodid=10412&qty=2 |

**Transformation:**

The following transformation extracts the list of query values from the URL. Note that the equals sign is included in the matching pattern so that you don't accidentally pick up numeric values from the non-parameter part of the URL:

| Transformation Name | `Extract matches into Array` |
|---|---|
| **Parameter: Column** | `cartURL` |
| **Parameter: Pattern matching elements in list** | `` `=[digit]+` `` |

The two query parameter values have been extracted into an array of values, including the equals sign, which must be removed:

| Transformation Name | `Replace text or pattern` |
|---|---|
| **Parameter: Column** | `cartURL` |
| **Parameter: Find** | `'='` |
| **Parameter: Replace with** | `''` |
| **Parameter: Match all occurrences** | `true` |

You can now unnest these values into separate columns:

| Transformation Name | `Unnest Objects into columns` |
|---|---|
| **Parameter: Column** | `extractlist_cartURL` |
| **Parameter: Paths to elements** | `'[0]','[1]'` |

After you rename the two columns to `prodId` and `Qty`, you can delete the column generated by the first transformation.

**Results:**

| Username | cartURL | prodId | Qty |
|---|---|---|---|
| joe.robinson | http://example123.com/cart.asp?prodid=1001&qty=2 | 1001 | 2 |
| steph.schmidt | http://example123.com/cart.asp?prodid=1005&qty=4 | 1005 | 4 |
| jack.holmes | http://example123.com/cart.asp?prodid=2102&qty=1 | 2102 | 1 |
| tina.jones | http://example123.com/cart.asp?prodid=10412&qty=2 | 10412 | 2 |

### Extract counts from a ragged array using extractlist

This example illustrates how to extract values from a column.

**Source:**

The following dataset contains counts of support emails processed by each member of the support team for individual customers over a six-month period. In this case, you are interested in the total number of emails processed for each customer.

Unfortunately, the data is ragged, as there are no entries for a support team member if he or she has not answered an email for a customer.

| custId | startDate | endDate | supportEmailCount |
|--------|-----------|---------|-------------------|
| C001 | 7/15/2015 | 12/31/2015 | ["Max":"2","Ted":"0","Sally":"12","Jack":"6","Sue":"4"] |
| C002 | 7/15/2015 | 12/31/2015 | ["Sally":"4","Sue":"3"] |
| C003 | 7/15/2015 | 12/31/2015 | ["Ted":"12","Sally":"2"] |
| C004 | 7/15/2015 | 12/31/2015 | ["Jack":"7","Sue":"4","Ted":"5"] |

If the data is imported from a CSV file, you might need to make some simple Replace Text or Pattern transformations to clean up the data to look like the above example.

**Transformation:**

Use the following transformation to extract just the numeric values from the `supportEmailCount` array:

| Transformation Name | Extract matches into Array |
|---------------------|----------------------------|
| Parameter: Column | supportEmailCount |
| Parameter: Pattern matching elements in list | `` `{digit}+` `` |

You should now have a column `extractlist_supportEmailCount` containing a ragged array. You can use the following transformations to convert this data to a comma-separated list of values:

| Transformation Name | Replace text or pattern |
|---------------------|-------------------------|
| Parameter: Column | extractlist_supportEmailCount |
| Parameter: Find | `` `[` `` |
| Parameter: Replace with | '' |
| Parameter: Match all occurrences | true |

| Transformation Name | Replace text or pattern |
|---------------------|-------------------------|
| Parameter: Column | extractlist_supportEmailCount |
| Parameter: Find | `` `]` `` |
| Parameter: Replace with | '' |
| Parameter: Match all occurrences | true |

| Transformation Name | Replace text or pattern |
|---------------------|-------------------------|
| Parameter: Column | extractlist_supportEmailCount |

| Parameter: Find | `` ` " ` `` |
|---|---|
| Parameter: Replace with | `' '` |
| Parameter: Match all occurrences | `true` |

Convert the column to String data type.

You can now split out the column into separate columns containing individual values in the modified source. The `limit` parameter specifies the number of splits to create, resulting in 5 new columns, which is the maximum number of entries in the source arrays.

| Transformation Name | `Split by delimiter` |
|---|---|
| Parameter: Column | `extractlist_supportEmailCount` |
| Parameter: Option | `On pattern` |
| Parameter: Match pattern | `','` |
| Parameter: Number of columns to create | `4` |

You might have to set the type for each generated column to Integer. If you try to use a New Formula transformation to calculate the sum of all of the generated columns, it only returns values for the first row because the missing rows are null values.

In the columns containing null values, select the missing value bar in the data histogram. Select the Replace suggestion card, and modify the transformation to write a `0` in place of the null value, as follows:

| Transformation Name | `Edit column with formula` |
|---|---|
| Parameter: Columns | `extractlist_supportEmailCount3` |
| Parameter: Formula | `'0'` |
| Parameter: Group rows by | `ismissing([extractlist_supportEmailCount3])` |

Repeat this step for any other column containing null values.

You can now use the following to sum the values in the generated columns:

| Transformation Name | `New formula` |
|---|---|
| Parameter: Formula type | `Single row formula` |
| Parameter: Formula | `(extractlist_supportEmailCount1 + extractlist_supportEmailCount2 + extractlist_supportEmailCount3 + extractlist_supportEmailCount4 + extractlist_supportEmailCount5)` |

**Results:**

After renaming the generated column to `totalSupportEmails` and dropping the columns used to create it, your dataset should look like the following:

| custId | startDate | endDate | supportEmailCount | totalSupportEmails |
|--------|-----------|---------|-------------------|--------------------|
| C001 | 7/15/2015 | 12/31/2015 | ["Max":"2","Ted":"0","Sally":"12","Jack":"6","Sue":"4"] | 24 |
| C002 | 7/15/2015 | 12/31/2015 | ["Sally":"4","Sue":"3"] | 7 |
| C003 | 7/15/2015 | 12/31/2015 | ["Ted":"12","Sally":"2"] | 14 |
| C004 | 7/15/2015 | 12/31/2015 | ["Jack":"7","Sue":"4","Ted":"5"] | 16 |

# Filter Transform

**Contents:**

---

> **NOTE:** Transforms are a part of the underlying language, which is not directly accessible to users. This content is maintained for reference purposes only. For more information on the user-accessible equivalent to transforms, see *Transformation Reference*.

Removes or keeps the matching rows of data, based on a condition that you specify or a custom formula that you apply.

## Basic Usage

You can filter your dataset based on the following condition types:

**Example - is missing**

```
filter missing: qty action: Drop
```

**Output:** Deletes all rows in which the value in the `qty` column is missing.

**Example - is mismatched**

```
filter col: CoName mismatched: 'String' action: Drop
```

**Output:** Deletes all rows in which the value in the `CoName` column does not match the String data type.

**Example - is exactly**

```
filter col: basic exactly: find(basic, '545', true, 1) == 8 action: Keep
```

**Output:** Keeps all rows where `545` appears at the eighth character in the `basic` column.

**Example - is one of**

---

```
filter col: zipCode oneOf: '94104','94105' action: Keep
```

**Output:** Keeps all rows in which the value of the `zipCode` column is either `94104` or `94105` and deletes all other rows in the dataset.

**Example - Less than (or equal to)**

```
filter col: row_number lessThanEqual: 5 action: Keep
```

**Output:** Keeps all rows in the dataset where the value in the `row_number` column is less than or equal to `5`. All other rows are deleted.

**Example - Greater than (or equal to)**

```
filter col: row_number greaterThanEqual: 10 action: Drop
```

**Output:** Deletes all rows in the dataset where the value in `row_number` is greater than or equal to `10`.

**Example - Is Between**

```
filter col: row_number greaterThan: 5 lessThanEqual: 15 action: Keep
```

**Output:** Keeps all rows where the `row_number` value is greater than `5` or less than or equal to `15`. All other rows are deleted.

**Example - Contains**

```
filter col: phoneNum contains: `\({digit}{3}\)` action: Keep
```

**Output:** Keeps all rows where the `phoneNum` value contains a three-digit pattern surrounded by parentheses (`XXX`). All other rows are deleted.

**Example - Starts with**

```
filter col: phoneNum startsWith: '(981)' action: Keep
```

**Output:** Keeps all rows where the `phoneNum` value begins with `(981)`. All other rows are deleted.

**Example - Ends with**

```
filter col: zipCode endsWith: `\-{digit}{4}` action: Drop
```

**Output:** Deletes all rows where the `zipCode` value ends with a four-digit extension.

**Example - custom formula**

```
filter row: (row_number >= 25 && firstName == 'Steve') action: Keep
```

**Output:** Keeps all rows where the `row_number` value is greater than or equal to 25 and the `firstName` value is `Steve`. All other rows are deleted.

## Syntax and Parameters

```
filter col:column_ref type: 'filter_str' [missing: column_ref] [exactly: expression_ref]
[mismatched: 'data_type_str'] [exactly: expression] [oneOf: 'string_1','string_2'] [lesstha
n | lessThanEqual: numVal] [greaterthan | greaterThanEqual: numVal] [contains:
string_or_pattern] [startsWith|endsWith: string_or_pattern] action: [Drop|Keep]
```

| Token | Required? | Data Type | Description |
|---|---|---|---|
| filter | Y | transform | Name of the transform |
| type | Y | string | String value representing the type of filtering to perform. |
| row | N | string | Expression identifying the row or rows to filter. If expression evaluates to `true` for a row, the row is either kept or deleted. |
| col | N | string | Name of the column or expression for columns to delete |
| missing | N | string | Name of column to evaluate for missing values. |
| mismatched | N | string | String literal for the data type to check for mismatches. |
| exactly | N | string | String literal, Pattern , or regular expression that evaluates to an exact match for a row value in the specified column. |
| oneOf | N | string | List of string literals, any of which can be matched. |
| lessThan or lessThanEqual | N | integer, decimal, or expression | Integer or decimal literal or expression evaluating to numeric value below which results in a match. Can also match on the specified expression exactly.<br><br>Parameter is also used for the Between condition type. |
| greaterThan or greaterThanEqual | N | integer, decimal, or expression | Integer or decimal literal or expression evaluating to numeric value above which results in a match. Can also match on the specified expression exactly.<br><br>Parameter is also used for the Between condition type. |
| contains | N | string | String literal, Pattern , or regular expression to be matches somewhere within the specified column values. |
| startsWith | N | string | String literal, Pattern , or regular expression to match the beginnings of the values in the specified column. |
| endsWith | N | string | String literal, Pattern , or regular expression to match the endings of the values in the specified column. |
| action | Y | string | `Drop` or `Keep` the listed columns |

For more information on syntax standards, see *Language Documentation Syntax Notes*.

**type**

String literal for the type of filtering to perform. For more information on these string literal values, see
*Valid Data Type Strings*.

**Usage Notes:**

| Required? | Data Type |
|---|---|
| Yes | String literal that corresponds to one of the supported data type values |

**row**

Expression to identify the row or rows on which to perform the transform. Expression must evaluate to `true` or `false`.

**Examples:**

| Expression | Description |
|---|---|
| `Score >= 50` | true if the value in the Score column is greater than 50. |
| `LEN(LastName) > 8` | true if the length of the value in the LastName column is greater than 8. |
| `ISMISSING([Title])` | true if the row value in the Title column is missing. |
| `ISMISMATCHED(Score,['Integer'])` | true if the row value in the Score column is mismatched against the Integer data type. |

**Example:**

```
delete row: (lastContactDate < 01/01/2010 || status == 'Inactive')
```

**Output:** Deletes any row in the dataset where the `lastContactDate` is before January 1, 2010 or the status is `Inactive`.

**Usage Notes:**

| Required? | Data Type |
|---|---|
| Yes | Expression that evaluates to `true` or `false` |

**col**

Identifies the column or columns to which to apply the transform. You can specify one column or more columns.

**Usage Notes:**

| Required? | Data Type |
|---|---|
| Yes | String (column name) |

**missing**

For the Is Missing condition type, this value specifies the column to check for missing values.

**Usage Notes:**

| Required? | Data Type |
|---|---|
| Required for Is Missing condition type only | String (column name) |

**mismatched**

For the Mismatched condition type, this value specifies string for the data type identifier value to check for mismatches. For more information, see *Valid Data Type Strings*.

The `col` parameter is also required.

**Usage Notes:**

| Required? | Data Type |
|---|---|
| Required for Mismatched condition type only | String (data type identifier) |

### exactly

For the Exactly condition type, this value is a String literal, Pattern , or regular expression that exactly matches row values in the specified column.

The `col` parameter is also required.

**Usage Notes:**

| Required? | Data Type |
|---|---|
| Required for Exactly condition type only | String (expression) |

### oneOf

For the One Of condition type, this value is a list of string literals,  Trifacta patterns, or regular expressions. If a row value for the specified column matches one of these expressions, the row is either deleted or kept.

The `col` parameter is also required.

**Usage Notes:**

| Required? | Data Type |
|---|---|
| Required for One Of condition type only | List of string literals, Patterns , or regular expressions |

### lessThan or lessThanEqual

For the Less Than conditional types, this value is an Integer or Decimal literal or an expression that evaluates to an Integer or Decimal literal. If the value in the specified column is less than (or optionally equal to) this value, then the row is either deleted or kept.

The `col` parameter is also required.

**Usage Notes:**

| Required? | Data Type |
|---|---|
| Required for Less than (or equal to) condition type only | Integer or Decimal literal or expression evaluating to one of these data types |

### greaterThan or greaterThanEqual

For the Less Than conditional types, this value is an Integer or Decimal literal or an expression that evaluates to an Integer or Decimal literal. If the value in the specified column is greater than (or optionally equal to) this value, then the row is either deleted or kept.

The `col` parameter is also required.

**Usage Notes:**

| Required? | Data Type |
|---|---|
| | |

| Required for Greater than (or equal to) condition type only | Integer or Decimal type or expression evaluating to one of these data types |
|---|---|

## contains

For the Contains condition type, this value identifies a String literal,  Pattern , or regular expression, which is used to evaluate partial or full matches to row values in the specified column.

The `col` parameter is also required.

**Usage Notes:**

| Required? | Data Type |
|---|---|
| Required for Contains condition type only | String literal, Pattern , or regular expression |

## startsWith

For the Starts With condition type, this value identifies the String literal,  Pattern , or regular expression with which a value must start in the specified column to match.

The `col` parameter is also required.

**Usage Notes:**

| Required? | Data Type |
|---|---|
| Required for Starts with condition type only | String literal, Pattern , or regular expression |

## endsWith

For the Ends With condition type, this value identifies the String literal,  Pattern , or regular expression with which a value must end in the specified column to match.

The `col` parameter is also required.

**Usage Notes:**

| Required? | Data Type |
|---|---|
| Required for Ends with condition type only | String literal, Pattern , or regular expression |

## action

Identifies whether the action performed by the transformation:

- `Drop` - Listed columns are deleted from the dataset.
- `Keep` - Listed columns are retained in the dataset, and all other columns are deleted.

**Usage Notes:**

| Required? | Data Type |
|---|---|
| Yes | String (`Drop` or `Keep`) |

## Examples

See above.

# Flatten Transform

**Contents:**

---

> **NOTE:** Transforms are a part of the underlying language, which is not directly accessible to users. This content is maintained for reference purposes only. For more information on the user-accessible equivalent to transforms, see *Transformation Reference.*

Unpacks array data into separate rows for each value. This transform operates on a single column.

This transform does not reference keys in the array. If your array data contains keys, use the `unnest` transform. See *Unnest Transform*.

## Basic Usage

```
flatten col: myArray
```

**Output:** Generates a separate row for each value in the array. Values of other columns in generated rows are copied from the source.

## Syntax and Parameters

```
flatten: col: column_ref
```

| Token | Required? | Data Type | Description |
|-------|-----------|-----------|-------------|
| flatten | Y | transform | Name of the transform |
| col | Y | string | Source column name |

For more information on syntax standards, see *Language Documentation Syntax Notes*.

**col**

Identifies the column to which to apply the transform. You can specify only one column.

**Usage Notes:**

| Required? | Data Type |
|-----------|-----------|
| Yes | String (column name) |

# Examples

> **Tip:** For additional examples, see *Common Tasks*.

### Example - Flatten an array

This section describes how to flatten the values in an Array into separate rows in your dataset.

**Source:**

In the following example dataset, students took the same test three times, and their scores were stored in any array in the `Scores` column.

| LastName | FirstName | Scores |
|----------|-----------|--------------|
| Adams | Allen | [81,87,83,79] |
| Burns | Bonnie | [98,94,92,85] |
| Cannon | Chris | [88,81,85,78] |

**Transformation:**

When the data is imported, you might have to re-type the `Scores` column as an array:

| Transformation Name | `Change column data type` |
|---------------------|---------------------------|
| **Parameter: Columns** | `Scores` |
| **Parameter: New type** | `Array` |

You can now flatten the `Scores` column data into separate rows:

| Transformation Name | `Expand Array into rows` |
|---------------------|--------------------------|
| **Parameter: Column** | `Scores` |

**Results:**

| LastName | FirstName | Scores |
|----------|-----------|--------|
| Adams | Allen | 81 |
| Adams | Allen | 87 |
| Adams | Allen | 83 |
| Adams | Allen | 79 |
| Burns | Bonnie | 98 |
| Burns | Bonnie | 94 |
| Burns | Bonnie | 92 |
| Burns | Bonnie | 85 |
| Cannon | Chris | 88 |
| Cannon | Chris | 81 |

| | | |
|---|---|---|
| Cannon | Chris | 85 |
| Cannon | Chris | 78 |

> **Tip:** You can use aggregation functions on the above data to complete values like average, minimum, and maximum scores. When these aggregation calculations are grouped by student, you can perform the calculations for each student.

### Example - Flatten and unnest together

While the above example nicely flattens out your data, there are two potential problems with the results:

- There is no identifier for each test. For example, Allen Adams' score of 87 cannot be associated with the specific test on which he recorded the score.
- There is no unique identifier for each row.

The following example addresses both of these issues. It also demonstrates differences between the `unnest` and the `flatten` transform, including how you use `unnest` to flatten array data based on specified keys.

- For more information, see *Unnest Transform*.

This example illustrates you to use the flatten and unnest transforms.

### Source:

You have the following data on student test scores. Scores on individual scores are stored in the `Scores` array, and you need to be able to track each test on a uniquely identifiable row. This example has two goals:

1. One row for each student test
2. Unique identifier for each student-score combination

| LastName | FirstName | Scores |
|---|---|---|
| Adams | Allen | [81,87,83,79] |
| Burns | Bonnie | [98,94,92,85] |
| Cannon | Charles | [88,81,85,78] |

### Transformation:

When the data is imported from CSV format, you must add a `header` transform and remove the quotes from the `Scores` column:

| Transformation Name | Rename column with row(s) |
|---|---|
| Parameter: Option | Use row(s) as column names |
| Parameter: Type | Use a single row to name columns |
| Parameter: Row number | 1 |

| Transformation Name | Replace text or pattern |
|---|---|
| Parameter: Column | colScores |
| Parameter: Find | '\"' |

| | |
|---|---|
| **Parameter: Replace with** | `' '` |
| **Parameter: Match all occurrences** | `true` |

**Validate test date:** To begin, you might want to check to see if you have the proper number of test scores for each student. You can use the following transform to calculate the difference between the expected number of elements in the `Scores` array (4) and the actual number:

| | |
|---|---|
| **Transformation Name** | `New formula` |
| **Parameter: Formula type** | `Single row formula` |
| **Parameter: Formula** | `(4 - arraylen(Scores))` |
| **Parameter: New column name** | `'numMissingTests'` |

When the transform is previewed, you can see in the sample dataset that all tests are included. You might or might not want to include this column in the final dataset, as you might identify missing tests when the recipe is run at scale.

**Unique row identifier:** The `Scores` array must be broken out into individual rows for each test. However, there is no unique identifier for the row to track individual tests. In theory, you could use the combination of `LastName-FirstName-Scores` values to do so, but if a student recorded the same score twice, your dataset has duplicate rows. In the following transform, you create a parallel array called `Tests`, which contains an index array for the number of values in the `Scores` column. Index values start at `0`:

| | |
|---|---|
| **Transformation Name** | `New formula` |
| **Parameter: Formula type** | `Single row formula` |
| **Parameter: Formula** | `range(0,arraylen(Scores))` |
| **Parameter: New column name** | `'Tests'` |

Also, we will want to create an identifier for the source row using the `sourcerownumber` function:

| | |
|---|---|
| **Transformation Name** | `New formula` |
| **Parameter: Formula type** | `Single row formula` |
| **Parameter: Formula** | `sourcerownumber()` |
| **Parameter: New column name** | `'orderIndex'` |

**One row for each student test:** Your data should look like the following:

| LastName | FirstName | Scores | Tests | orderIndex |
|---|---|---|---|---|
| Adams | Allen | [81,87,83,79] | [0,1,2,3] | 2 |
| Burns | Bonnie | [98,94,92,85] | [0,1,2,3] | 3 |
| Cannon | Charles | [88,81,85,78] | [0,1,2,3] | 4 |

Now, you want to bring together the `Tests` and `Scores` arrays into a single nested array using the `arrayzip` function:

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | `Single row formula` |
| Parameter: Formula | `arrayzip([Tests,Scores])` |

Your dataset has been changed:

| LastName | FirstName | Scores | Tests | orderIndex | column1 |
|---|---|---|---|---|---|
| Adams | Allen | [81,87,83,79] | [0,1,2,3] | 2 | [[0,81],[1,87],[2,83],[3,79]] |
| Adams | Bonnie | [98,94,92,85] | [0,1,2,3] | 3 | [[0,98],[1,94],[2,92],[3,85]] |
| Cannon | Charles | [88,81,85,78] | [0,1,2,3] | 4 | [[0,88],[1,81],[2,85],[3,78]] |

Use the following to unpack the nested array:

| Transformation Name | Expand arrays to rows |
|---|---|
| Parameter: Column | `column1` |

Each test-score combination is now broken out into a separate row. The nested Test-Score combinations must be broken out into separate columns using the following:

| Transformation Name | Unnest Objects into columns |
|---|---|
| Parameter: Column | `column1` |
| Parameter: Paths to elements | `'[0]','[1]'` |

After you delete `column1`, which is no longer needed you should rename the two generated columns:

| Transformation Name | Rename columns |
|---|---|
| Parameter: Option | `Manual rename` |
| Parameter: Column | `column_0` |
| Parameter: New column name | `'TestNum'` |

| Transformation Name | Rename columns |
|---|---|
| Parameter: Option | `Manual rename` |
| Parameter: Column | `column_1` |
| Parameter: New column name | `'TestScore'` |

**Unique row identifier:** You can do one more step to create unique test identifiers, which identify the specific test for each student. The following uses the original row identifier `OrderIndex` as an identifier for the student and the `TestNumber` value to create the `TestId` column value:

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | `Single row formula` |
| | |

| Parameter: Formula | `(orderIndex * 10) + TestNum` |
|---|---|
| Parameter: New column name | `'TestId'` |

The above are integer values. To make your identifiers look prettier, you might add the following:

| Transformation Name | `Merge columns` |
|---|---|
| Parameter: Columns | `'TestId00','TestId'` |

**Extending:** You might want to generate some summary statistical information on this dataset. For example, you might be interested in calculating each student's average test score. This step requires figuring out how to properly group the test values. In this case, you cannot group by the `LastName` value, and when executed at scale, there might be collisions between first names when this recipe is run at scale. So, you might need to create a kind of primary key using the following:

| Transformation Name | `Merge columns` |
|---|---|
| Parameter: Columns | `'LastName','FirstName'` |
| Parameter: Separator | `'-'` |
| Parameter: New column name | `'studentId'` |

You can now use this as a grouping parameter for your calculation:

| Transformation Name | `New formula` |
|---|---|
| Parameter: Formula type | `Single row formula` |
| Parameter: Formula | `average(TestScore)` |
| Parameter: Group rows by | `studentId` |
| Parameter: New column name | `'avg_TestScore'` |

**Results:**

After you delete unnecessary columns and move your columns around, the dataset should look like the following:

| TestId | LastName | FirstName | TestNum | TestScore | studentId | avg_TestScore |
|---|---|---|---|---|---|---|
| TestId0021 | Adams | Allen | 0 | 81 | Adams-Allen | 82.5 |
| TestId0022 | Adams | Allen | 1 | 87 | Adams-Allen | 82.5 |
| TestId0023 | Adams | Allen | 2 | 83 | Adams-Allen | 82.5 |
| TestId0024 | Adams | Allen | 3 | 79 | Adams-Allen | 82.5 |
| TestId0031 | Adams | Bonnie | 0 | 98 | Adams-Bonnie | 92.25 |
| TestId0032 | Adams | Bonnie | 1 | 94 | Adams-Bonnie | 92.25 |
| TestId0033 | Adams | Bonnie | 2 | 92 | Adams-Bonnie | 92.25 |
| TestId0034 | Adams | Bonnie | 3 | 85 | Adams-Bonnie | 92.25 |
| TestId0041 | Cannon | Chris | 0 | 88 | Cannon-Chris | 83 |
| TestId0042 | Cannon | Chris | 1 | 81 | Cannon-Chris | 83 |

| TestId0043 | Cannon | Chris | 2 | 85 | Cannon-Chris | 83 |
| TestId0044 | Cannon | Chris | 3 | 78 | Cannon-Chris | 83 |

# Header Transform

> **NOTE:** Transforms are a part of the underlying language, which is not directly accessible to users. This content is maintained for reference purposes only. For more information on the user-accessible equivalent to transforms, see *Transformation Reference*.

Uses one row from the dataset sample as the header row for the table. Each value in this row becomes the name of the column in which it is located.

This transform might be automatically applied as one of the first steps of your recipe.

> **NOTE:** If source row number information is not available due to changes in the dataset, this transform may not be available.

> **NOTE:** Each column in any row that is part of a column header in a dataset with parameters should have a valid value that is consistent with corresponding values across all files in the dataset. If your files have missing or empty values in rows that are used as headers in your recipe, these rows may be treated as data rows during the import process, which may result in unexpected or missing column values. For more information, see *Create Dataset with Parameters*.

## Basic Usage

```
header sourcerownumber: 4
```

**Output:** The values from Row #4 of the original dataset are used, if available, as the names for each column. If the row is not available, the specified row data can be retrieved, and the transform fails.

## Syntax and Parameters

```
header sourcerownumber: row_num
```

| Token | Required? | Data Type | Description |
|---|---|---|---|
| header | Y | transform | Name of the transform |
| sourcerownumber | Y | integer (positive) | Row number from the original data to use as the header. |

For more information on syntax standards, see *Language Documentation Syntax Notes*.

**sourcerownumber**

The `sourcerownumber` parameter defines the row number to apply to the transform step.

This parameter references the original row number of the sample in the dataset.

- `sourceownumber` parameter must be an integer that is less than or equal to the total number of rows in the original sample.
- If the corresponding row has been deleted from the dataset, the transform step generates an error.

**Example:**

```
header sourcerownumber: 4
```

**Output:** Uses row #4 from the source row numbers of the sample as the header the columns.

**Usage Notes:**

| Required? | Data Type |
|---|---|
| Yes | integer (positive) |

## Examples

> **Tip:** For additional examples, see *Common Tasks*.

**Example - Header from row that is not the first one**

This example illustrates how you can rename columns based on the contents of specified rows.

**Source:**

You have imported the following racer data on heat times from a CSV file. When loaded in the Transformer page, it looks like the following:

| (rowId) | column2 | column3 | column4 | column5 |
|---|---|---|---|---|
| 1 | Racer | Heat 1 | Heat 2 | Heat 3 |
| 2 | Racer X | 37.22 | 38.22 | 37.61 |
| 3 | Racer Y | 41.33 | DQ | 38.04 |
| 4 | Racer Z | 39.27 | 39.04 | 38.85 |

In the above, the `(rowId)` column references the row numbers displayed in the data grid; it is not part of the dataset. This information is available when you hover over the black dot on the left side of the screen.

**Transformation:**

You have examined the best performance in each heat according to the sample. You then notice that the data contains headers, but you forget how it was originally sorted. The data now looks like the following:

| (rowId) | column2 | column3 | column4 | column5 |
|---|---|---|---|---|
| 1 | Racer Y | 41.33 | DQ | 38.04 |
| 2 | Racer | Heat 1 | Heat 2 | Heat 3 |
| 3 | Racer X | 37.22 | 38.22 | 37.61 |
| 4 | Racer Z | 39.27 | 39.04 | 38.85 |

You can use the following transformation to use the third row as your header for each column:

| Transformation Name | Rename column with row(s) |
|---|---|
| Parameter: Option | Use row(s) as column names |
| Parameter: Type | Use a single row to name columns |
| Parameter: Row number | 3 |

**Results:**

After you have applied the above transformation, your data should look like the following:

| (rowId) | Racer | Heat_1 | Heat_2 | Heat_3 |
|---|---|---|---|---|
| 3 | Racer Y | 41.33 | DQ | 38.04 |
| 2 | Racer X | 37.22 | 38.22 | 37.61 |
| 4 | Racer Z | 39.27 | 39.04 | 38.85 |

# Keep Transform

> **NOTE:** Transforms are a part of the underlying language, which is not directly accessible to users. This content is maintained for reference purposes only. For more information on the user-accessible equivalent to transforms, see *Transformation Reference*.

Retains a set of rows in your dataset, which are specified by the conditional in the `row` expression. All other rows are removed from the dataset.

The `keep` transform is the opposite of the `delete` transform. See *Delete Transform*.

## Basic Usage

```
keep row:(customerStatus == 'active')
```

**Output:** For each row in the dataset, if the value of the `customerStatus` column is `active`, then the row is retained. Otherwise, the row is deleted from the dataset.

## Syntax and Parameters

```
keep row:(expression)
```

| Token | Required? | Data Type | Description |
|-------|-----------|-----------|-------------|
| keep | Y | transform | Name of the transform |
| row | Y | string | Expression identifying the row or rows to keep. If expression evaluates to `true` for a row, the row is retained. |

For more information on syntax standards, see *Language Documentation Syntax Notes*.

### row

Expression to identify the row or rows on which to perform the transform. Expression must evaluate to `true` or `false`.

**Examples:**

| Expression | Description |
|------------|-------------|
| `Score >= 50` | `true` if the value in the `Score` column is greater than 50. |
| `LEN(LastName) > 8` | `true` if the length of the value in the `LastName` column is greater than 8. |
| `ISMISSING([Title])` | `true` if the row value in the `Title` column is missing. |
| `ISMISMATCHED(Score,['Integer'])` | `true` if the row value in the `Score` column is mismatched against the Integer data type. |

For the `keep` transform, if the expression for the `row` parameter evaluates to `true` for a row, it is kept in the dataset. Otherwise, it is removed.

**Example:**

```
keep row: (lastOrder >= 10000 && status == 'Active')
```

**Output:** Retains all rows in the dataset where the `lastOrder` value is greater than or equal to 10,000 and the customer status is `Active`.

**Usage Notes:**

| Required? | Data Type |
|---|---|
| Yes | Expression that evaluates to `true` or `false` |

## Examples

> **Tip:** For additional examples, see *Common Tasks*.

**Example - Remove old products and keep new orders**

This examples illustrates how you can keep and delete rows from your dataset.

**Source:**

Your dataset includes the following order information. You want to edit your dataset so that:

- All orders for products that are no longer available are removed. These include the following product IDs: `P 100`, `P101`, `P102`, `P103`.
- All orders that were placed within the last 90 days are retained.

| OrderId | OrderDate | ProdId | ProductName | ProductColor | Qty | OrderValue |
|---|---|---|---|---|---|---|
| 1001 | 6/14/2015 | P100 | Hat | Brown | 1 | 90 |
| 1002 | 1/15/2016 | P101 | Hat | Black | 2 | 180 |
| 1003 | 11/11/2015 | P103 | Sweater | Black | 3 | 255 |
| 1004 | 8/6/2015 | P105 | Cardigan | Red | 4 | 320 |
| 1005 | 7/29/2015 | P103 | Sweeter | Black | 5 | 375 |
| 1006 | 12/1/2015 | P102 | Pants | White | 6 | 420 |
| 1007 | 12/28/2015 | P107 | T-shirt | White | 7 | 390 |
| 1008 | 1/15/2016 | P105 | Cardigan | Red | 8 | 420 |
| 1009 | 1/31/2016 | P108 | Coat | Navy | 9 | 495 |

**Transformation:**

First, you remove the orders for old products. Since the set of products is relatively small, you can start first by adding the following:

> **NOTE:** Just preview this transformation. Do not add it to your recipe yet.

| Transformation Name | `Filter rows` |
|---|---|
| **Parameter: Condition** | `Custom formula` |
| **Parameter: Type of formula** | `Custom single` |
| **Parameter: Condition** | `(ProdId == 'P100')` |
| **Parameter: Action** | `Delete matching rows` |

When this step is previewed, you should notice that the top row in the above table is highlighted for removal. Notice how the transformation relies on the `ProdId` value. If you look at the `ProductName` value, you might notice that there is a misspelling in one of the affected rows, so that column is not a good one for comparison purposes.

You can add the other product IDs to the transformation in the following expansion of the transformation, in which any row that has a matching `ProdId` value is removed:

| Transformation Name | `Filter rows` |
|---|---|
| **Parameter: Condition** | `Custom formula` |
| **Parameter: Type of formula** | `Custom single` |
| **Parameter: Condition** | `(ProdId == 'P100' \|\| ProdId == 'P101' \|\| ProdId == 'P102' \|\| ProdId == 'P103')` |
| **Parameter: Action** | `Delete matching rows` |

When the above step is added to your recipe, you should see data that looks like the following:

| OrderId | OrderDate | ProdId | ProductName | ProductColor | Qty | OrderValue |
|---|---|---|---|---|---|---|
| 1004 | 8/6/2015 | P105 | Cardigan | Red | 4 | 320 |
| 1007 | 12/28/2015 | P107 | T-shirt | White | 7 | 390 |
| 1008 | 1/15/2016 | P105 | Cardigan | Red | 8 | 420 |
| 1009 | 1/31/2016 | P108 | Coat | Navy | 9 | 495 |

Now, you can filter out of the dataset orders that are older than 90 days. First, add a column with today's date:

| Transformation Name | `New formula` |
|---|---|
| **Parameter: Formula type** | `Single row formula` |
| **Parameter: Formula** | `'2/25/16'` |
| **Parameter: New column name** | `'today'` |

Keep the rows that are within 90 days of this date using the following:

| Transformation Name | Filter rows |
|---|---|
| Parameter: Condition | `Custom formula` |
| Parameter: Type of formula | `Custom single` |
| Parameter: Condition | `datedif(OrderDate,today,day) <= 90` |
| Parameter: Action | `Keep matching rows` |

Don't forget to delete the `today` column, which is no longer needed:

| Transformation Name | `Delete columns` |
|---|---|
| Parameter: Columns | `today` |
| Parameter: Action | `Delete selected columns` |

**Results:**

| OrderId | OrderDate | ProdId | ProductName | ProductColor | Qty | OrderValue |
|---|---|---|---|---|---|---|
| 1007 | 12/28/2015 | P107 | T-shirt | White | 7 | 390 |
| 1008 | 1/15/2016 | P105 | Cardigan | Red | 8 | 420 |
| 1009 | 1/31/2016 | P108 | Coat | Navy | 9 | 495 |

# Merge Transform

**Contents:**

---

> **NOTE:** Transforms are a part of the underlying language, which is not directly accessible to users. This content is maintained for reference purposes only. For more information on the user-accessible equivalent to transforms, see *Transformation Reference*.

Merges two or more columns in your dataset to create a new column of String type. Optionally, you can insert a delimiter between the merged values.

> **NOTE:** This transform applies to String columns or other columns that can be interpreted as strings (for example, Zip codes could be interpreted as five-digit strings). To concatenate arrays, use the `ARRAYCONCAT` function. See *ARRAYCONCAT Function*.

## Basic Usage

**Column example:**

```
merge col:Column1,Column2 as:'MergedCol'
```

**Output:** Merges the contents of `Column1` and `Column2` in that order into a new column called `MergedCol`.

**Column and string literal example:**

```
merge col:'PID',ProdId with:'-'
```

**Output:** Merges the string `PID` and the values in `ProdId` together. The string and the value are separated by a dash. Example output value: `PID-00123`.

## Syntax and Parameters

```
merge col:column_ref [with:string_literal_pattern] [as:'new_column_name']
```

| Token | Required? | Data Type | Description |
|-------|-----------|-----------|-------------|
| merge | Y | transform | Name of the transform |
| col | Y | string | Source column name or names |
| with | N | string | String literal used in the new column as a separator between the merged column values |
| as | N | string | Name of the newly generated column |

For more information on syntax standards, see *Language Documentation Syntax Notes*.

**col**

Identifies columns or range of columns as source data for the transform. You must specify multiple columns.

To specify multiple columns:

- Discrete column names are comma-separated.
- Values for column names are case-sensitive.

```
merge col: Prefix,Root,Suffix
```

**Output:** Merges the columns `Prefix`, `Root`, and `Suffix` in that order into a new column.

**Usage Notes:**

| Required? | Data Type |
|-----------|-----------|
| Yes | String (column name) |

**with**

**Merge Columns transformation:** Specifies the delimiter between columns that are merged. If this parameter is not specified, no delimiter is applied.

**Replace Text or Pattern transformation:** Specifies the replacement value.

```
merge col: CustId,ProdId with:'-'
```

**Output:** Merges the columns `CustId` and `ProdId` into a new column with a dash (`-`) between the source values in the new column.

**Usage Notes:**

| Required? | Data Type |
|-----------|-----------|
| No | String (column name) |

**as**

Name of the new column that is being generated. If the `as` parameter is not specified, a default name is used.

```
merge col: CustId,ProdId with:'-' as:'PrimaryKey'
```

**Output:** Merges the columns `CustId` and `ProdId` into a new column with a dash (`-`) between the source values in the new column. New column is named, `PrimaryKey`.

**Usage Notes:**

| Required? | Data Type |
|-----------|-----------|
| No | String (column name) |

# Examples

## Example - Merging date values

You have date information stored in multiple columns. You can merge columns together to form a single date value.

**Source:**

| OrderId | Month | Day | Year |
|---------|-------|-----|------|
| 1001 | 2 | 14 | 2008 |
| 1002 | 7 | 22 | 2009 |
| 1003 | 11 | 22 | 2010 |
| 1004 | 12 | 25 | 2011 |

**Transformation:**

```
merge col:Month~Year with:'/' as:'Date'
```

**Results:**

When you add the transform and move the generated `Date` column, your dataset should look like the following. Note that the generated column is automatically inferred as Datetime values.

| OrderId | Month | Day | Year | Date |
|---------|-------|-----|------|------|
| 1001 | 2 | 14 | 2008 | 2/14/2008 |
| 1002 | 7 | 22 | 2009 | 7/22/2009 |
| 1003 | 11 | 22 | 2010 | 11/22/2010 |
| 1004 | 12 | 25 | 2011 | 12/25/2011 |

## Example - Use merge and settype to clean up numeric data that should be treated as other data types

This example illustrates how to clean up data by changing its data type to String, manipulating it using String functions, and then retyping the data to its proper data type.

**Functions:**

| Item | Description |
|------|-------------|
| IF Function | The `IF` function allows you to build if/then/else conditional logic within your transforms. |
| LEN Function | Returns the number of characters in a specified string. String value can be a column reference or string literal. |
| MERGE Function | Merges two or more columns of String type to generate output of String type. Optionally, you can insert a delimiter between the merged values. |

**Source:**

The following example contains customer ID and Zip code information in two columns. When this data is loaded into the Transformer page, it is initially interpreted as numeric, since it contains all numerals.

The four-digit `ZipCode` values should have five digits, with a `0` in front.

| CustId | ZipCode |
|---|---|
| 4020123 | 1234 |
| 2012121 | 94105 |
| 3212012 | 94101 |
| 1301212 | 2020 |

**Transformation:**

**CustId column:** This column needs to be retyped as String values. You can set the column data type to String through the column drop-down, which is rendered as the following transformation:

| | |
|---|---|
| Transformation Name | `Change column data type` |
| Parameter: Columns | `CustId` |
| Parameter: New type | `String` |

While the column is now of String type, future transformations might cause it to be re-inferred as Integer values. To protect against this possibility, you might want to add a marker at the front of the string. This marker should be removed prior to execution.

The basic method is to create a new column containing the customer ID marker (`C`) and then merge this column and the existing `CustId` column together. It's useful to add such an indicator to the front in case the customer identifier is a numeric value that could be confused with other numeric values. Also, this merge step forces the value to be interpreted as a String value, which is more appropriate for an identifier.

| | |
|---|---|
| Transformation Name | `Merge columns` |
| Parameter: Columns | `'C',CustId` |

You can now delete the `CustId` columns and rename the new column as `CustId`.

**ZipCode column:** This column needs to be converted to valid Zip Code values. For ease of use, this column should be of type String:

| | |
|---|---|
| Transformation Name | `Change column data type` |
| Parameter: Columns | `ZipCode` |
| Parameter: New type | `Zipcode` |

The transformation below changes the value in the `ZipCode` column if the length of the value is four in any row. The new value is the original value prepended with the numeral `0`:

| | |
|---|---|
| Transformation Name | `Edit column with formula` |
| Parameter: Columns | `ZipCode` |
| Parameter: Formula | `if(len($col) == 4, merge(['0',$col]), $col)` |

This column might now be re-typed as Zipcode type.

**Results:**

| CustId | ZipCode |
|----------|---------|
| C4020123 | 01234 |
| C2012121 | 94105 |
| C3212012 | 94101 |
| C1301212 | 02020 |

Remember to remove the `C` marker from the `CustId` column. Select the `C` value in the `CustId` column and choose the `replace` transform. You might need to re-type the cleaned data as String data.

# Move Transform

**Contents:**

---

> **NOTE:** Transforms are a part of the underlying language, which is not directly accessible to users. This content is maintained for reference purposes only. For more information on the user-accessible equivalent to transforms, see *Transformation Reference.*

Moves the specified column or columns before or after another column in your dataset.

## Basic Usage

```
move col: FirstName before: LastName
```

**Output:** The column `FirstName` is moved before the column `LastName`.

## Syntax and Parameters

```
move col:column_ref before:column_ref | after:column_ref
```

> **NOTE:** At least one of the `after` or `before` parameters must be included.

| Token | Required? | Data Type | Description |
|---|---|---|---|
| move | Y | transform | Name of the transform |
| col | Y | string | Name of column or columns to move |
| after | `before` or `after` is required. | string | Name of column after which to place the moved columns |
| before | `before` or `after` is required. | string | Name of column before which to place the moved columns |

For more information on syntax standards, see *Language Documentation Syntax Notes.*

**col**

Identifies the column or columns to which to apply the transform. You can specify one or more columns.

To specify multiple columns:

- Discrete column names are comma-separated.
- Values for column names are case-sensitive.

**Usage Notes:**

| Required? | Data Type |
|-----------|-----------|
| Yes | String (column name) |

**after**

```
move col: ZipCode after: State
```

**Output:** Moves column `ZipCode` so that it appears after the column `State`.

The column after which the object of the transform is to be placed.

**Usage Notes:**

| Required? | Data Type |
|-----------|-----------|
| Either `after` or `before` is required. Do not include both. | Column reference |

**before**

```
move col: ItemName before: ItemColor
```

**Output:** Moves the column `ItemName` so that it appears before the `ItemColor` column in your dataset.

The column before which the object of the transform is to be placed.

**Usage Notes:**

| Required? | Data Type |
|-----------|-----------|
| Either `after` or `before` is required. Do not include both. | Column reference |

## Examples

> **Tip:** For additional examples, see *Common Tasks*.

**Example - Reorder Columns**

**Source:**

| C | B | A |
|---|---|---|
| 7 | 4 | 1 |
| 8 | 5 | 2 |
| 9 | 6 | 3 |

**Transformation:**

| | |
|---|---|
| **Transformation Name** | `Move columns` |
| **Parameter: Column(s)** | `A` |
| **Parameter: Option** | `Before` |
| **Parameter: Column** | `C` |

| | |
|---|---|
| **Transformation Name** | `Move columns` |
| **Parameter: Column(s)** | `C` |
| **Parameter: Option** | `After` |
| **Parameter: Column** | `B` |

**Results:**

| A | B | C |
|---|---|---|
| 1 | 4 | 7 |
| 2 | 5 | 8 |
| 3 | 6 | 9 |

### Example - Move multiple columns

The following examples illustrate how you can move multiple columns in a single transformation.

**Source:**

| A | B | C | D | E | F |
|---|---|---|---|---|---|
| 1 | 2 | 3 | 4 | 5 | 6 |
| 7 | 8 | 9 | 10 | 11 | 12 |
| 13 | 14 | 15 | 16 | 17 | 18 |
| 19 | 20 | 21 | 22 | 23 | 24 |

**Transformation:**

| | |
|---|---|
| **Transformation Name** | `Move columns` |
| **Parameter: Column(s)** | `A,B,C` |
| **Parameter: Option** | `After` |
| **Parameter: Column** | `E` |

| | |
|---|---|
| **Transformation Name** | `Move columns` |
| **Parameter: Column(s)** | `B,F` |
| **Parameter: Option** | `Before` |
| **Parameter: Column** | `E` |

**Results:**

| D | B | F | E | A | C |
|---|---|---|---|---|---|
| 4 | 2 | 6 | 5 | 1 | 3 |
| 10 | 8 | 12 | 11 | 7 | 9 |
| 16 | 14 | 18 | 17 | 13 | 15 |
| 22 | 20 | 24 | 23 | 19 | 21 |

# Nest Transform

**Contents:**

- *Basic Usage*
- *Syntax and Parameters*
  - *col*
  - *into*
  - *as*
- *Examples*

---

> **NOTE:** Transforms are a part of the underlying language, which is not directly accessible to users. This content is maintained for reference purposes only. For more information on the user-accessible equivalent to transforms, see *Transformation Reference.*

Creates an Object or Array of values using column names and their values as key-value pairs for one or more columns. Generated column type is determined by the `into` parameter.

The `nest` transform is the opposite of `unnest`, which unpacks Object data into separate columns and rows. See *Unnest Transform*.

## Basic Usage

| ItemA | ItemB |
|-------|-------|
| 22    | 33    |
| 44    | 55    |

**Object example:**

```
nest col:ItemA,ItemB into:'obj' as:'myObj'
```

**Output:** See below.

| ItemA | ItemB | myObj |
|-------|-------|-------|
| 22    | 33    | {"ItemA":"22","ItemB","33"} |
| 44    | 55    | {"ItemA":"44","ItemB","55"} |

**Array example:**

```
nest col:ItemA,ItemB into:'array' as:'myArray'
```

**Output:** Output arrays do not include the column name.

| ItemA | ItemB | myArray |
|-------|-------|---------|
| 22    | 33    | ["22","33"] |
| 44    | 55    | ["44","55"] |

## Syntax and Parameters

```
nest col:column_ref [into: object|array] [as:'new_column_name']
```

| Token | Required? | Data Type | Description |
|-------|-----------|-----------|-------------|
| nest | Y | transform | Name of the transform |
| col | Y | string | Source column name |
| into | N | string | Data type of output column: `object` (default) or `array` |
| as | N | string | Name of newly generated column |

For more information on syntax standards, see *Language Documentation Syntax Notes*.

**col**

Identifies the column or columns to which to apply the transform. You can specify one column or more columns.

To specify multiple columns:

- Discrete column names are comma-separated.
- Values for column names are case-sensitive.

For each listed column, a new pair of key and value columns is generated.

```
nest col: Qty, Amount
```

**Output:** Builds an Object of the data from the columns `Qty` and `Amount` .

**Usage Notes:**

| Required? | Data Type |
|-----------|-----------|
| Yes | String (column name) |

**into**

Defines the output column type. Accepted values:

- `object`
- `array`

If this parameter is not specified, the output type is Object.

**Usage Notes:**

| Required? | Data Type |
|-----------|-----------|
| No (Object is default) | String (data type name) |

**as**

Name of the new column that is being generated. If the `as` parameter is not specified, a default name is used.

```
nest col: CustId,ProdId as:'masterNest'
```

**Output:** Nests the data from the columns `CustId` and `ProdId` into a new column called, `masterNest`.

**Usage Notes:**

| Required? | Data Type |
|-----------|-----------|
| No | String (column name) |

## Examples

> **Tip:** For additional examples, see *Common Tasks*.

**Source:**

In the following example, furniture product dimensions are stored in separate columns in cm.

| Category | Name | Length_cm | Width_cm | Height_cm |
|----------|------|-----------|----------|-----------|
| bench | Hooska | 118.11 | 74.93 | 46.34 |
| lamp | Tansk | 30.48 | 30.48 | 165.1 |
| bookshelf | Brock | 27.94 | 160.02 | 201.93 |
| couch | Loafy | 95 | 227 | 83 |

**Transformation:**

Use the `nest` transform to bundle the data into a single column.

| Transformation Name | Nest columns into Objects |
|---------------------|---------------------------|
| Parameter: Columns | Length_cm,Width_cm,Height_cm |
| Parameter: Nest columns to | Array |
| Parameter: New column name | 'Dimensions' |

**Results:**

| Category | Name | Length_cm | Width_cm | Height_cm | Dimensions |
|----------|------|-----------|----------|-----------|------------|
| bench | Hooska | 118.11 | 74.93 | 46.34 | {"Length_cm":"118.11","Width_cm":"74.93","Height_cm":"46.34"} |
| lamp | Tansk | 30.48 | 30.48 | 165.1 | {"Length_cm":"30.48","Width_cm":"30.48","Height_cm":"165.1"} |
| bookshelf | Brock | 27.94 | 160.02 | 201.93 | {"Length_cm":"27.94","Width_cm":"160.02","Height_cm":"201.93"} |
| couch | Loafy | 95 | 227 | 83 | {"Length_cm":"95,"Width_cm":"227","Height_cm":"83"} |

# Pivot Transform

**Contents:**

---

> **NOTE:** Transforms are a part of the underlying language, which is not directly accessible to users. This content is maintained for reference purposes only. For more information on the user-accessible equivalent to transforms, see *Transformation Reference.*

When you pivot data, the values from the selected column or columns become new columns in the dataset, each of which contains a summary calculation that you specify.

- This calculation can be based on all rows for totals across the dataset or based on group of rows you define in the transform.
- When your data has been pivoted, the unused columns are dropped, leaving only the operative data for faster evaluation.
- Transform accepts one or more columns as inputs to the pivot.

> **Tip:** This transform is very useful in Preview mode for quick discovery and analysis.

## Basic Usage

```
pivot col: Dates value:SUM(Sales) group: prodId
```

**Output:** Reshapes your dataset to include a `ProdId` column and new columns for each distinct value in the `Dat es` column. Each distinct `ProdId` value is represented by a separate row in the reshaped dataset. Within each row, the new columns contain a sum of sales for the product for each date in the dataset.

## Syntax and Parameters

```
pivot col:column_ref value: FUNCTION(arg1,arg2) [group: group_col] [limit:int_num]
```

| Token | Required? | Data Type | Description |
|-------|-----------|-----------|-------------|
| pivot | Y | transform | Name of the transform |
| col | N | string | Source column name or names |
| value | N | string | Expression containing the aggregation function or functions used to pivot the source column data |
| group | N | string | Column name or names containing the values by which to group for calculation |

| | | | | |
|---|---|---|---|---|
| limit | N | integer (positive) | Maximum number of unique values in a group. Default is 50. | |

For more information on syntax standards, see *Language Documentation Syntax Notes*.

### col

Identifies the column or columns to which to apply the transform.

**Single-column example:**

The specified column contains the values that become new columns in the dataset.

```
pivot col: autoBrand value:AVERAGE(autoPrice) group:State
```

**Output:** Reshapes the dataset to contain the `State` column followed by columns for each distinct value in the `au toBrand` column. Each value in those columns is the average value in the `autoPrice` column, as grouped by `St ate` value.

**Multi-column example:**

For multiple input columns, the transform generates new columns for each combination of the inputs. See the example below.

**Usage Notes:**

| Required? | Data Type |
|---|---|
| No | String (column name) |

### value

For this transform, the `value` parameter contains the aggregation function and its parameters, which define the set of rows to which the function is applied.

> **NOTE:** For the `value` parameter, you can only use aggregation functions. Nested functions are not supported. For more information, see *Aggregate Functions* .

**Usage Notes:**

| Required? | Data Type |
|---|---|
| No | String containing a list of aggregation functions each containing one column reference |

### group

For this transform, this parameter specifies the column whose values are used to group the dataset prior to applying the specified function. You can specify multiple columns as comma-separated values.

> **NOTE:** Transforms that use the `group` parameter can result in non-deterministic re-ordering in the data grid. However, you should apply the `group` parameter, particularly on larger datasets, or your job may run out of memory and fail.  To enforce row ordering, you can use the `sort` transform. For more information, see *Sort Transform*.

**Usage Notes:**

| Required? | Data Type |
|---|---|
| No | String (column name) |

## limit

The `limit` parameter defines the maximum number of unique values permitted in a group. If it is not specified, the default value for this parameter is `50`.

> **NOTE:** Be careful setting this parameter too high. In some cases, the application can run out of memory generating the results, and your results can fail.

### Usage Notes:

| Required? | Data Type |
|---|---|
| No. Default value is `50`. | Integer (positive) |

# Examples

> **Tip:** For additional examples, see *Common Tasks*.

### Example - Basic Pivot

**Source:**

The following dataset contains information about sales across one weekend and two states for three different products.

| Date | State | ProdId | Sales |
|---|---|---|---|
| 3/9/16 | CA | Big Trike 9000 | 500 |
| 3/9/16 | NV | Fast GoKart 5000 | 200 |
| 3/9/16 | CA | SuperQuick Scooter | 700 |
| 3/9/16 | CA | Fast GoKart 5000 | 900 |
| 3/9/16 | NV | Big Trike 9000 | 300 |
| 3/9/16 | NV | SuperQuick Scooter | 250 |
| 3/10/16 | NV | Fast GoKart 5000 | 50 |
| 3/10/16 | NV | Big Trike 9000 | 400 |
| 3/10/16 | NV | SuperQuick Scooter | 150 |
| 3/10/16 | CA | Big Trike 9000 | 600 |
| 3/10/16 | CA | SuperQuick Scooter | 800 |
| 3/10/16 | CA | Fast GoKart 5000 | 1100 |

**Transformation 1:** Sum of sales by State for each Date

Apply this transformation in Preview only, just so you can see the results:

| Transformation Name | Pivot columns |
|---|---|
| Parameter: Column labels | Date |
| Parameter: Row labels | State |
| Parameter: Values | sum(Sales) |

Cancel the transformation update.

| State | sum_Sales_03/09/2016 | sum_Sales_03/10/2016 |
|---|---|---|
| CA | 2100 | 2500 |
| NV | 750 | 600 |

**Transform 2:** Sum of Sales by Date for each State

| Transformation Name | Pivot columns |
|---|---|
| Parameter: Column labels | State |
| Parameter: Row labels | Date |
| Parameter: Values | sum(Sales) |

| Date | sum_Sales_CA | sum_Sales_NV |
|---|---|---|
| 03/09/2016 | 2100 | 750 |
| 03/10/2016 | 2500 | 600 |

Cancel the transformation update again.

**Transform 3:** Sum of Sales by product ID for each State

| Transformation Name | Pivot columns |
|---|---|
| Parameter: Column labels | State |
| Parameter: Row labels | ProdId |
| Parameter: Values | sum(Sales) |

| ProdId | sum_Sales_CA | sum_Sales_NV |
|---|---|---|
| Big Trike 9000 | 1100 | 700 |
| Fast GoKart 5000 | 2000 | 500 |
| SuperQuick Scooter | 1500 | 300 |

### Example - Multi-column Pivot

The Pivot Columns transformation supports the ability to use the values in multiple columns to specify the columns that are generated. When two or more columns are used in the pivot, columns containing values of all possible combinations from the source columns are generated.

In the following source data, sales data on individual products is organized by brand, product name, and month:

| Brand | Product | Month | Sales |
|---|---|---|---|
| AAA | Towels | January | 95 |
| AAA | Napkins | January | 113 |
| B | Towels | January | 99 |
| B | Tissues | January | 88 |
| AAA | Towels | February | 108 |
| AAA | Napkins | February | 91 |
| B | Towels | February | 85 |
| B | Tissues | February | 105 |
| AAA | Towels | March | 81 |
| AAA | Napkins | March | 92 |
| B | Towels | March | 112 |
| B | Tissues | March | 104 |

### Transformation:

If you wanted to create summary sales information for each product by month, you might choose to create a `pivot` on the `Product` column. However, if you look at the column values, you might notice that the `Paper towels` product is available for both the `AAA` and `B` brands. In this case, you must perform a multi-column pivot on these two columns, like the following:

| Transformation Name | Pivot columns |
|---|---|
| Parameter: Column labels | Brand,Product |
| Parameter: Row labels | Month |
| Parameter: Values | average(Sales) |

### Results:

| Month | avg_Sales_B_Towels | avg_Sales_B_Tissues | avg_Sales_AAA_Napkins | avg_Sales_AAA_Towels |
|---|---|---|---|---|
| January | 99 | 88 | 113 | 95 |
| February | 85 | 105 | 91 | 108 |
| March | 112 | 104 | 92 | 81 |

### Example - Aggregate Values

You can use the `pivot` transform to perform aggregation calculations on values in a column.

### Source:

In the following table, you can review test results from three different tests for 10 students:

| StudentId | TestId | Score |
|---|---|---|
| s001 | t001 | 98 |
| s001 | t002 | 98 |
| s001 | t003 | 87 |
| s002 | t001 | 92 |

| | | |
|---|---|---|
| s002 | t002 | 96 |
| s002 | t003 | 79 |
| s003 | t001 | 99 |
| s003 | t002 | 76 |
| s003 | t003 | 94 |
| s004 | t001 | 93 |
| s004 | t002 | 99 |
| s004 | t003 | 80 |
| s005 | t001 | 79 |
| s005 | t002 | 80 |
| s005 | t003 | 84 |
| s006 | t001 | 93 |
| s006 | t002 | 74 |
| s006 | t003 | 89 |
| s007 | t001 | 86 |
| s007 | t002 | 81 |
| s007 | t003 | 97 |
| s008 | t001 | 82 |
| s008 | t002 | 73 |
| s008 | t003 | 79 |
| s009 | t001 | 96 |
| s009 | t002 | 97 |
| s009 | t003 | 79 |
| s010 | t001 | 80 |
| s010 | t002 | 79 |
| s010 | t003 | 75 |

**Transformation:**

For each student, you're interested in a student's average score and maximum score. You create the following pivot transform, which groups computations of average score and maximum score columns by studentId value.

| | |
|---|---|
| **Transformation Name** | Pivot columns |
| **Parameter: Row labels** | StudentId |
| **Parameter: Values** | average(Score),max(Score) |
| **Parameter: Max number of columns to create** | 50 |

For the generated average column, you want results rounded to two decimal places:

| | |
|---|---|
| **Transformation Name** | Edit column with formula |

| Parameter: Columns | average_Score |
|---|---|
| Parameter: Formula | round(average_Score, 2) |

**Results:**

| StudentId | average_Score | max_Score |
|---|---|---|
| s001 | 94.33 | 98 |
| s002 | 89 | 96 |
| s003 | 89.67 | 99 |
| s004 | 90.67 | 99 |
| s005 | 81 | 84 |
| s006 | 85.33 | 93 |
| s007 | 88 | 97 |
| s008 | 78 | 82 |
| s009 | 90.67 | 97 |
| s010 | 78 | 80 |

# Rename Transform

**Contents:**

---

> **NOTE:** Transforms are a part of the underlying language, which is not directly accessible to users. This content is maintained for reference purposes only. For more information on the user-accessible equivalent to transforms, see *Transformation Reference*.

Renames one or more columns based on specified names, patterns, row values, or prefixes and suffixes. You can also rename to uppercase or lowercase values.

> **NOTE:** Column names are case-insensitive and cannot begin with whitespace.

> **Tip:** To prevent potential issues with downstream systems, you should limit your column lengths to no more than 128 characters.

Other ways to rename:

- It's easier to rename columns through the user interface.
    - To rename a single column, double-click the column name or select **Rename...** from the column drop-down in the Transformer Page.
    - To rename multiple columns, you can select values in the Column Browser and perform batch renames.
- Transforms that generate new columns might support the `as` parameter, which enables specifying the name of the new column. Using the `as` parameter avoids the extra step of adding a `rename` transform after column generation.
    - See *Derive Transform*.
    - See *Extractkv Transform*.

- See *Extractlist Transform*.
- See *Merge Transform*.
- See *Nest Transform*.

## Basic Usage

**Rename a single column manually:**

```
rename mapping: [oldName,'NewName']
```

**Output:** Renames the column `OldName` to `NewName`.

**Rename multiple columns:**

This transform supports multiple methods for renaming two or more columns in a single step. See below for examples.

## Syntax and Parameters

```
rename: type: renameType col: column_ref [mapping: [column1,'newColumn1Name'], [column2,'ne
wColumn2Name']] [prefix: 'strPrefix'] [suffix: 'strSuffix'] [keepIndex: NumOfChars] [on:
`patternOrLiteral`] [with: 'replacementString'] [method: strMethodType] [sourcerownumber:
intRowNum] [sourcerownumbers: strCommaList]
```

| Token | Required? | Data Type | Description |
|---|---|---|---|
| rename | Y | transform | Name of the transform |
| type | Y | string | Enum of supported renaming types. For more information, see "type" below. |
| col | Y | string | Name of column or columns to rename |
| prefix | N | string | (type=prefix) Prefix to prepend to the column name |
| suffix | N | string | (type=suffix) Suffix to append to the column name |
| keepIndex | N | integer | (type=keepLeft or type=keepRight) Number of characters on the left or right side of the column to retain |
| mapping | N | array | (type=manual) Array containing mappings from old column name and new column name |
| on | N | string | (type=findAndReplace) Pattern or string literal for which to search each column name |
| with | N | string | (type=findAndReplace) Replacement value for found pattern or string literal in column names |
| method | N | string | (type=header) Enum of supported methods for renaming headers based on row numbers. See "method" below. |
| sourcerownumber | N | integer | (type=header,method=index) Row number from the source data to use as the new names for the selected columns |
| sourcerownumbers | N | string | (type=header,method=multi) Comma-separated list of row numbers from the source data to use as the new names for the selected columns |
| separator | N | string | (type=header,method=multi) String to separate row values when multiple rows are used to define column headers. |
| filltype | N | boolean | (type=header,method=multi) When `true`, empty row values are replaced with the nearest row value to the left in the replacement column header. |
| sanitize | N | boolean | When `true`, column names are sanitized after they have been renamed. |

For more information on syntax standards, see *Language Documentation Syntax Notes*.

## type

Type of column rename to perform. Options:

| Type | Description | Other required parameters |
|---|---|---|
| manual | Manual rename of one or more columns. | `mapping` - array specifies one or more manual column renames. |
| prefix | Rename column by adding a prefix to it. | `prefix` - string value with which to prepend the column name. |
| suffix | Rename column by adding a suffix to it. | `suffix` - string value with which to append the column name. |
| findAndReplace | Rename the column using find and replace. | `on` - value to find in column names<br><br>`with` - value to replace found value in column names. |
| keepLeft | Keep the leftmost characters in the column name | `keepIndex` - number of characters on the left side of the column name to keep |
| keepRight | Keep the right characters in the column name | `keepIndex` - number of characters on the right side of the column name to keep |
| upper | Rename column to use all uppercase characters. | None. |
| lower | Rename column to use all lowercase characters. | None. |
| header | Rename column based on a row number | `method` - enum defining the method of rename of the column headers.<br><br>Additional parameters are required depending on the `method` value. See below. |

**Usage Notes:**

| Required? | Data Type |
|---|---|
| Yes | Enum of supported String values. |

## col

Identifies the column or columns to which to apply the transform.

> **Tip:** The `col` parameter must be specified for all types. You can use the * wildcard to apply to all columns.

**Usage Notes:**

| Required? | Data Type |
|---|---|
| Yes | Comma-separated list of column names (String values) |

## mapping

An array describing the old names and new names for each column to rename.

Example:

| Old column name | New column name |
|---|---|
| | |

| | |
|---|---|
| column1 | FirstName |
| column2 | LastName |
| column3 | Phone |

Transform step:

```
rename mapping: [column1,'FirstName'],[column2,'LastName'],[column3,'Phone']
```

**Usage Notes:**

| Required? | Data Type |
|---|---|
| No | Array |

### prefix

For batch rename using prefixes, this parameter specifies the string value with which to precede each of the column names.

**Usage Notes:**

| Required? | Data Type |
|---|---|
| No | String |

### suffix

For batch rename using suffixes, this parameter specifies the string value with which to append each of the column names.

**Usage Notes:**

| Required? | Data Type |
|---|---|
| No | String |

### on

For batch rename using find and replace, this parameter specifies the pattern or string literal to use to match values.

Replacement values are specified with the `with` parameter.

**Usage Notes:**

| Required? | Data Type |
|---|---|
| No | String (pattern or literal) |

### with

For batch rename using find and replace, this parameter specifies the literal string values with which to replace the found pattern.

Find patterns and values are specified with the `on` parameter.

**Usage Notes:**

| Required? | Data Type |
|-----------|-----------|
| No | String (pattern or literal) |

### keepIndex

For batch rename using keep-left or keep-right, this parameter specifies the number of characters on the left or right side of the column name to retain as the new column name.

> **NOTE:** This value must be an integer greater than 0. If this value results in multiple columns having the same new name, you must specify a greater value to create unique names or use a different rename method.

**Usage Notes:**

| Required? | Data Type |
|-----------|-----------|
| No | Integer |

### method

Type of row-based column rename to perform. Options:

| Type | Description | Other required parameters |
|------|-------------|---------------------------|
| index | Rename based on a single row number. | `sourcerownumber` - source row number values to use as new column headers. |
| filter | Use the values in the first row in the current sample to use as the new column names. | None. |
| multi | Rename column by adding a suffix to it. | `sourcerownumbers` - comma-separated list of values for the source row numbers to use as new column headers. Other parameters are applicable. See below. |

**Usage Notes:**

| Required? | Data Type |
|-----------|-----------|
| No | Enum of supported row-based rename types (String). |

### sourcerownumber

The row number from the original source data which contains the values to use to rename all columns in the dataset. The row is removed from its original position.

> **NOTE:** If source row number information is no longer available, this method cannot be used for column rename.

**Usage Notes:**

| Required? | Data Type |
| --- | --- |
| No | Integer (Positive value) |

### sourcerownumbers

A comma-separated list of the row numbers from the original source data containing the values to use to rename all columns in the dataset. These rows are removed from its original position.

> **NOTE:** If source row number information is no longer available, this method cannot be used for column rename.

**Usage Notes:**

| Required? | Data Type |
| --- | --- |
| No | Comma-separated list of row numbers (String) |

### separator

String value to separate row value entries in the column header when multiple rows are used to rename columns. This value is not required.

**Usage Notes:**

| Required? | Data Type |
| --- | --- |
| No | String |

### filltype

When set to `true`, empty values encountered in row values used to rename the column header are replaced using the nearest non-empty column value to the left. Default is `false`.

**Usage Notes:**

| Required? | Data Type |
| --- | --- |
| No | Boolean |

### sanitize

When set to `true`, new column headers are sanitized to remove special characters. Default is `false`.

**Usage Notes:**

| Required? | Data Type |
| --- | --- |
| No | Boolean |

## Examples

> **Tip:** For additional examples, see *Common Tasks*.

## Rename a column

In the following dataset, the length columns do not include any units of measure.

> **Tip:** For downstream consumption, any column that contains a measure should include the units of measure in the column name. Avoid including units of measure in cell values, which forces the column to be retyped as String type.

**Source:**

| Object | LengthX | LengthY | LengthZ |
|--------|---------|---------|---------|
| ObjA   | 10      | 20      | 30      |
| ObjB   | 3       | 4       | 5       |
| ObjC   | 6       | 9       | 12      |

**Transformation:**

Perhaps you know the units are centimeters. You can rename using the following:

| Transformation Name | Rename columns |
|---------------------|----------------|
| Parameter: Option | Manual rename |
| Parameter: Column | LengthX |
| Parameter: New column name | 'LengthX_cm' |

Now, you want to convert the units of measure to inches. You can use the New Formula transformation to convert values and generate a new column name:

| Transformation Name | New formula |
|---------------------|-------------|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | (LengthX_cm * 0.393701) |
| Parameter: New column name | 'LengthX_in' |

You might want to reformat the generated values using transformations like the following, which rounds the results to two decimal points:

| Transformation Name | Edit column with formula |
|---------------------|--------------------------|
| Parameter: Columns | LengthX_in |
| Parameter: Formula | numformat(LengthX_in, '##.00') |

Repeat the above steps for the other length columns.

**Results:**

| Object | LengthX_cm | LengthY_cm | LengthZ_cm | LengthX_in | LengthY_in | LengthZ_in |
|--------|------------|------------|------------|------------|------------|------------|

| ObjA | 10 | 20 | 30 | 3.94 | 7.87 | 11.81 |
|------|----|----|----|------|------|-------|
| ObjB | 3  | 4  | 5  | 1.18 | 1.57 | 1.97  |
| ObjC | 6  | 9  | 12 | 2.36 | 3.54 | 4.72  |

You can delete the original columns if needed.

### Rename multiple columns

#### Source:

| column1 | column2 | column3 | column4 | column5 |
|---------|---------|---------|---------|---------|
| data1   | data2   | data3   | data4   | data5   |

#### Transformation:

Add prefix:

| Transformation Name | `Rename columns` |
|---------------------|------------------|
| **Parameter: Option** | `Add prefix` |
| **Parameter: Columns** | `column1,column2,column3,column4,column5` |
| **Parameter: Prefix** | `'new_'` |

| new_column1 | new_column2 | new_column3 | new_column4 | new_column5 |
|-------------|-------------|-------------|-------------|-------------|
| data1       | data2       | data3       | data4       | data5       |

Add suffix:

| Transformation Name | `Rename columns` |
|---------------------|------------------|
| **Parameter: Option** | `Add suffix` |
| **Parameter: Columns** | `new_column1,new_column2,new_column3,new_column4,`<br>`new_column5` |
| **Parameter: Suffix** | `'a'` |

| new_column1a | new_column2a | new_column3a | new_column4a | new_column5a |
|--------------|--------------|--------------|--------------|--------------|
| data1        | data2        | data3        | data4        | data5        |

Convert to UPPERCASE:

| Transformation Name | `Rename columns` |
|---------------------|------------------|
| **Parameter: Option** | `Convert to UPPERCASE` |
| **Parameter: Columns** | `Column1,Column2,Column3,Column4,Column5` |

| COLUMN1 | COLUMN2 | COLUMN3 | COLUMN4 | COLUMN5 |
|---------|---------|---------|---------|---------|
| data1   | data2   | data3   | data4   | data5   |

Find and replace:

| Transformation Name | Rename columns |
|---|---|
| Parameter: Option | Find and replace |
| Parameter: Columns | new_column1,new_column2,new_column3,new_column4, new_column5 |
| Parameter: Find | '_column' |
| Parameter: Replace | '_field' |

| new_field1 | new_field2 | new_field3 | new_field4 | new_field5 |
|---|---|---|---|---|
| data1 | data2 | data3 | data4 | data5 |

# Replace Transform

**Contents:**

---

> **NOTE:** Transforms are a part of the underlying language, which is not directly accessible to users. This content is maintained for reference purposes only. For more information on the user-accessible equivalent to transforms, see *Transformation Reference*.

Replaces values within the specified column or columns based on the string literal, pattern, or location within the cell value, as specified in the transform.

The `replace` transform is used primarily to match on patterns within a string. For entire cell replacement across all rows of the column, use the `set` transform. See *Set Transform*.

## Basic Usage

**`on` parameter example:**

Specifies the string literal or pattern to match.

```
replace col: text on: 'honda' with:'toyota' global: true
```

**Output:** Replaces all instances in the `text` column of `honda` with `toyota`. If `honda` appears twice a cell, both instances are replaced with `toyota`.

**`at` parameter example:**

Specifies the beginning character and ending character as index values for the match.

```
replace col: text at: 2,6 with:'replacement text'
```

**Output:** For all values in the `text` column, replace the substring between character 2 and 6 in the column with the value `replacement text`. If the length of the original cell value is less than 6, the replacement value is inserted.

## Syntax and Parameters

```
replace col:column_ref with:'literal_replacement' [at:(start_index,end_index)] [on:
string_literal_pattern] [global:true|false]
```

| Token | Required? | Data Type | Description |
|-------|-----------|-----------|-------------|
| replace | Y | transform | Name of the transform |
| col | Y | string | Name of column where to make replacements |
| with | Y | see below | Literal value with which to replace matched values |
| after | N | string | String literal or pattern that precedes the pattern to match |
| at | N | Array | Two-integer array identifying the character indexes of start and end characters to match |
| before | N | string | String literal or pattern that appears after the pattern to match |
| from | N | string | String literal or pattern that identifies the start of the pattern to match |
| to | N | string | String literal or pattern that identifies the end of the pattern to match |
| on | N | string | String literal or pattern that identifies the cell characters to replace |
| global | N | boolean | If `true`, all occurrences of matches are replaced. Default is `false`. |

For more information on syntax standards, see *Language Documentation Syntax Notes*.

**col**

Identifies the column or columns to which to apply the transform. You can specify one or more columns.

To specify multiple columns:

- Discrete column names are comma-separated.
- Values for column names are case-sensitive.

```
replace col: MyCol on: 'MyString' with: 'myNewString'
```

**Output:** Replaces value `MyString` in `MyCol` column with `myNewString`.

**Usage Notes:**

| Required? | Data Type |
|-----------|-----------|
| Yes | String (column name) |

**with**

**Merge Columns transformation:** Specifies the delimiter between columns that are merged. If this parameter is not specified, no delimiter is applied.

**Replace Text or Pattern transformation:** Specifies the replacement value.

For the `replace` transform, this value must be a literal value. You can apply values of String or other data types. After replacement, the column data type is re-inferred.

> **NOTE:** Some regular expression capture groups with references (such as $2) are supported across all running environments. See *Capture Group References*.

**Usage Notes:**

| Required? | Data Type |
|-----------|-----------|

| | |
|---|---|
| Yes | Literal of any data type |

**after**

```
replace col:Primary_URL with:'' after:`http({any}|):`
```

**Output:** All content after the protocol identifier (`http:` or `https:`) is dropped.

A pattern identifier that precedes the value or pattern to match. Define the `after` parameter value using string literals, regular expressions, or Patterns .

**Usage Notes:**

| Required? | Data Type |
|---|---|
| No | String (string literal or pattern) |

- The `after` and `from` parameters are very similar. `from` includes the matching value as part of the replaced string.
- `after` can be used with either `to`, `on`, or `before`. See  *Pattern Clause Position Matching*.

**at**

```
replace col: MyCol at: 2,6 with:'MyNewString'
```

**Output:** Replace contents of `MyCol` that starts at the second character in the column and extends to the sixth character with the value `MyNewString`.

Identifies the start and end point of the pattern to interest.

Parameter inputs are in the form of `x,y` where `x` and `y` are positive integers indicating the starting character and ending character, respectively, of the pattern of interest.

- `x` must be less than `y`.
- If `y` is greater than the length of the value, the pattern is defined to the end of the value, and a match is made.

**Usage Notes:**

| Required? | Data Type |
|---|---|
| Must use either `on` or `at` parameter | Array of two Integers ( `X`,`Y` ) |

- For more information, see  *Pattern Clause Position Matching*.

**before**

A pattern identifier that occurs after the value or pattern to match. Define the pattern using string literals, regular expressions, or Patterns .

```
replace col:credit_card with:'****-***-***-' after:`{start}` before:`({digit}{4}){end}`
```

**Output:**

- Replaces first three groups of digits in the `credit_card` column with asterisks, effectively masking the number.

**Usage Notes:**

| Required? | Data Type |
|---|---|
| No | String or pattern |

- The `before` and `to` parameters are very similar. `to` includes the matching value as part of the replaced string.
- `before` can be used with either `from`, `on`, or `after`. See  *Pattern Clause Position Matching*.

### from

Identifies the pattern that marks the beginning of the value to match. It can be a string literal, Pattern , or regular expression. The `from` value is included in the match.

```
replace col: MyCol from: '<END>' with: ''
```

**Output:**

- All content from the string `<END>` to the end of the string value in `MyCol` is removed.

**Usage Notes:**

| Required? | Data Type |
|---|---|
| No | String or pattern |

- The `after` and `from` parameters are very similar. `from` includes the matching value as part of the replaced string.
- `from` can be used with either `to` or `before`. See  *Pattern Clause Position Matching*.

### to

Identifies the pattern that marks the ending of the value to match. Pattern can be a string literal,  Patterns , or regular expression. The `to` value is included in the match.

```
replace col:ssn with:'***-**-' to:`{digit}{3}-{digit}{2}-`
```

**Output:**

- Replace first two number groups in the column `ssn` with asterisks to mask the data.

**Usage Notes:**

| Required? | Data Type |
|---|---|
| No | String or pattern |

- The `before` and `to` parameters are very similar. `to` includes the matching value as part of the replaced string.
- `to` can be used with either `from` or `after`. See *Pattern Clause Position Matching*.

### on

```
replace col: MyCol on: `###ERROR` with:'No error here'
```

Identifies the pattern to match, which can be a string literal, Pattern , or regular expression.

**Usage Notes:**

| Required? | Data Type |
|---|---|
| Must use either `on` or `at` parameter | String or pattern |

- For more information, s ee *Pattern Clause Position Matching*.

**global**

Indicates whether any match should be applied to one instance or to all.

- (Default) If `false`, apply transform only to the first instance.
- If `true`, apply to all found matches.

**NOTE:** If you have specified the pattern to match with the `at` parameter, then the number of possible replacement instances is only 1, so the `global` parameter is not used.

**Usage Notes:**

| Required? | Data Type |
|---|---|
| No. Default is `false`. | Boolean |

## Examples

**Tip:** For additional examples, see *Common Tasks*.

**Example - Clean up marketing contact data with replace, set, and extract**

This example illustrates the different uses of the replacement transformations to replace or extract cell data.

**Source:**

The following dataset contains contact information that has been gathered by your marketing platform from actions taken by visitors on your website. You must clean up this data and prepare it for use in an analytics platform.

| LeadId | LastName | FirstName | Title | Phone | Request |
|---|---|---|---|---|---|
| LE160301001 | Jones | Charles | Chief Technical Officer | 415-555-1212 | reg |
| LE160301002 | Lyons | Edward | | 415-012-3456 | download whitepaper |
| LE160301003 | Martin | Mary | CEO | 510-555-5555 | delete account |
| LE160301004 | Smith | Talia | Engineer | 510-123-4567 | free trial |

**Transformation:**

**Title column:** For example, you first notice that some data is missing. Your analytics platform recognizes the string value, "`#MISSING#`" as an indicator of a missing value. So, you click the missing values bar in the Title column. Then, you select the Replace suggestion card. Note that the default replacement is a null value, so you click **Edit** and update it:

| | |
|---|---|
| **Transformation Name** | `Edit column with formula` |
| **Parameter: Columns** | `Title` |
| **Parameter: Formula** | `if(ismissing([Title]),'#MISSING#',Title)` |

**Request column:** In the Request column, you notice that the `reg` entry should be cleaned up. Add the following transformation, which replaces that value:

| | |
|---|---|
| **Transformation Name** | `Replace text or pattern` |
| **Parameter: Column** | `Request` |
| **Parameter: Find** | `` `{start}reg{end}` `` |
| **Parameter: Replace with** | `Registration` |

The above transformation uses a Pattern as the expression of the `on:` parameter. This expression indicates to match from the start of the cell value, the string literal `reg`, and then the end of the cell value, which matches on complete cell values of `reg` only.

This transformation works great on the sample, but what happens if the value is `Reg` with a capital `R`? That value might not be replaced. To improve the transformation, you can modify the transformation with the following Pattern in the `on` parameter, which captures differences in capitalization:

| | |
|---|---|
| **Transformation Name** | `Replace text or pattern` |
| **Parameter: Column** | `Request` |
| **Parameter: Find** | `` `{start}{[R|r]}eg{end}` `` |
| **Parameter: Replace with** | `'Registration'` |

Add the above transformation to your recipe. Then, it occurs to you that all of the values in the `Request` column should be capitalized in title or proper case:

| | |
|---|---|
| **Transformation Name** | `Edit column with formula` |
| **Parameter: Columns** | `Request` |
| **Parameter: Formula** | `proper(Request)` |

Now, all values are capitalized as titles.

**Phone column:** You might have noticed some issues with the values in the `Phone` column. In the United States, the prefix `555` is only used for gathering information; these are invalid phone numbers.

In the data grid, you select the first instance of `555` in the column. However, it selects all instances of that pattern, including ones that you don't want to modify. In this case, continue your selection by selecting the similar instance of `555` in the other row. In the suggestion cards, you click the Replace Text or Pattern transformation.

Notice, however, that the default Replace Text or Pattern transformation has also highlighted the second `555` pattern in one instance, which could be a problem in other phone numbers not displayed in the sample. You must modify the selection pattern for this transformation. In the `on:` parameter below, the Pattern has been modified to match only the instances of `555` that appear in the second segment in the phone number format:

| Transformation Name | Replace text or pattern |
| --- | --- |
| Parameter: Column | Phone |
| Parameter: Find | `` `{start}%{3}-555-%*{end}` `` |
| Parameter: Replace with | '#INVALID#' |
| Parameter: Match all occurrences | true |

Note the wildcard construct has been added (`%*`). While it might be possible to add a pattern that matches on the last four characters exactly (`%{4}`), that matching pattern would not capture the possibility of a phone number having an extension at the end of it. The above expression does.

> **NOTE:** The above transformation creates values that are mismatched with the Phone Number data type. In this example, however, these mismatches are understood to be for the benefit of the system consuming your Trifacta output.

**LeadId column:** You might have noticed that the lead identifier column (`LeadId`) contains some embedded information: a date value and an identifier for the instance within the day. The following steps can be used to break out this information. The first one creates a separate working column with this information, which allows us to preserve the original, unmodified column:

| Transformation Name | New formula |
| --- | --- |
| Parameter: Formula type | Single row formula |
| Parameter: Formula | LeadId |
| Parameter: New column name | 'LeadIdworking' |

You can now work off of this column to create your new ones. First, you can use the following replace transformation to remove the leading two characters, which are not required for the new columns:

| Transformation Name | Replace text or pattern |
| --- | --- |
| Parameter: Column | LeadIdworking |
| Parameter: Find | 'LE' |
| Parameter: Replace with | '' |

Notice that the date information is now neatly contained in the first characters of the working column. Use the following to extract these values to a new column:

| Transformation Name | Extract text or pattern |
| --- | --- |

| Parameter: Column to extract from | `LeadIdworking` |
|---|---|
| Parameter: Option | `Custom text or pattern` |
| Parameter: Text to extract | `` `{start}%{6}` `` |

The new `LeadIdworking2` column now contains only the date information. Cleaning up this column requires reformatting the data, retyping it as a Datetime type, and then applying the `dateformat` function to format it to your satisfaction. These steps are left as a separate exercise.

For now, let's just rename the column:

| Transformation Name | `Rename columns` |
|---|---|
| Parameter: Option | `Manual rename` |
| Parameter: Column | `LeadIdworking1` |
| Parameter: New column name | `'LeadIdDate'` |

In the first working column, you can now remove the date information using the following:

| Transformation Name | `Replace text or pattern` |
|---|---|
| Parameter: Column | `LeadIdworking` |
| Parameter: Find | `` `{start}%{6}` `` |
| Parameter: Replace with | `''` |

You can rename this column to indicate it is a daily identifier:

| Transformation Name | `Rename columns` |
|---|---|
| Parameter: Option | `Manual rename` |
| Parameter: Column | `LeadIdworking` |
| Parameter: New column name | `'LeadIdDaily'` |

### Results:

| LeadId | LeadIdDaily | LeadIdDate | LastName | FirstName | Title | Phone | Request |
|---|---|---|---|---|---|---|---|
| LE160301001 | 001 | 160301 | Jones | Charles | Chief Technical Officer | #INVALID# | Registration |
| LE160301002 | 002 | 160301 | Lyons | Edward | #MISSING# | 415-012-3456 | Download Whitepaper |
| LE160301003 | 003 | 160301 | Martin | Mary | CEO | #INVALID# | Delete Account |
| LE160301004 | 004 | 160301 | Smith | Talia | Engineer | 510-123-4567 | Free Trial |

**Example - Using capture group references for replacements**

The `replace` transform can take advantage of capture groups defined in the  Patterns  and regular expressions used to search for values within a column. A **capture group** is a sub-pattern within your pattern that defines a value that you can reference in the replacement.

> **NOTE:** For this transform, capture groups can be specified in the `on` parameter only.

In the following example, the `on` parameter defines two capture groups, and the `with` parameter references them in the replacement. In this example, any entry in the `camel_case` column that contains a lower-case letter followed immediately by an upper-case letter is replaced by the same value with a space inserted in the middle. The `$1` value references the first capture group in the corresponding  Pattern :

| Transformation Name | Replace text or pattern |
|---|---|
| **Parameter: Column** | `camel_case` |
| **Parameter: Find** | `` `({lower})({upper})` `` |
| **Parameter: Replace with** | `'$1 $2'` |
| **Parameter: Match all occurrences** | `true` |

| Capture Group | Description | Replacement Reference |
|---|---|---|
| `({lower})` | A single lower-case letter | `$1` |
| `({upper})` | A single upper-case letter | `$2` |

# Set Transform

**Contents:**

---

> **NOTE:** Transforms are a part of the underlying language, which is not directly accessible to users. This content is maintained for reference purposes only. For more information on the user-accessible equivalent to transforms, see *Transformation Reference*.

Replaces values in the specified column or columns with the specified value, which can be a literal or an expression. Expressions can use conditional functions to filter the set of rows.

The `set` transform is used to replace entire cell values. For replacement of partial cell values using literals or patterns, use the `replace` transform. See *Replace Transform*.

## Basic Usage

**Literal example:**

```
set col: Country value: 'USA'
```

**Output:** Sets the values of all rows in the `Country` column to `USA`.

**Multi-column Literal example:**

```
set col: SSN,Phone value: '##REDACTED###'
```

**Output:** Sets the values of all rows in the `SSN` and `Phone` columns to `##REDACTED##`.

**Expression example:**

```
set col: isAmerica value: IF(Country == 'USA', true', 'false')
```

**Output:** If the value in the `Country` column is `USA`, then the value in `isAmerica` is set to `true`.

**Placeholder example:**

You can substitute a placeholder value for the column name, which is useful if you are applying the same function across multiple columns. For example:

```
set col:score1,score2 value:IF ($col == 0, AVERAGE($col), $col)
```

**Output:** In the above transform, the values in `score1` and `score2` are set to the average of the column value when the value in the column is `0`. Note that the computation of average is applied across all rows in the column, instead of just the filtered rows.

**Window function example:**

You can use window functions in your `set` transforms:

```
set col: avgSales value: ROLLINGAVERAGE(POS_Sales, 7, 0) group: saleDate order: saleDate
```

**Output:** Calculate the value in the column of `avgSales` to be the rolling average of the `POS_Sales` values for the preceding seven days, grouped and ordered by the `saleDate` column. For more information, see *Window Functions*.

## Syntax and Parameters

```
set col:col1,[col2] value:(expression) [group: group_col]
```

| Token | Required? | Data Type | Description |
|-------|-----------|-----------|-------------|
| set | Y | transform | Name of the transform |
| col1 | Y | string | Column name |
| col2 | N | string | Column name |
| value | Y | string | Expression that generates the value to store in the column |
| group | N | string | If you are using aggregate or window functions, you can specify a `group` expression to identify the subset of records to apply the `value` expression. |

For more information on syntax standards, see *Language Documentation Syntax Notes*.

**col1, col2**

Identifies the column and optional additional columns to which to apply the transform.

```
set col: MyCol value: 'myNewString'
```

**Output:** Sets value in `MyCol` column to `myNewString`.

**Usage Notes:**

| Required? | Data Type |
|-----------|-----------|
| Yes | String (column name) |

**value**

Identifies the expression that is applied by the transform. The `value` parameter can be one of the following types:

- test predicates that evaluate to Boolean values (`value: myAge == '30'` yields a `true` or `false` value), or
- computational expressions ( `value: abs(pow(myCol,3))` ).

The expected type of `value` expression is determined by the transform type. Each type of expression can contain combinations of the following:

- **literal values:** `value: 'Hello, world'`
- **column references:** `value: amountOwed * 10`
- **functions:** `value: left(myString, 4)`
- **combinations:** `value: abs(pow(myCol,3))`

The types of any generated values are re-inferred by the platform.

**Usage Notes:**

| Required? | Data Type |
|-----------|-----------|
| Yes | String (literal, column name, or expression) |

### group

Identifies the column by which the dataset is grouped for purposes of applying the transform.

> **NOTE:** Transforms that use the `group` parameter can result in non-deterministic re-ordering in the data grid. However, you should apply the `group` parameter, particularly on larger datasets, or your job may run out of memory and fail. To enforce row ordering, you can use the `sort` transform. For more information, see *Sort Transform*.

If the value parameter contains aggregate or window functions, you can apply the `group` parameter to specify subsets of records across which the value computation is applied.

You can specify one or more columns by which to group using comma-separated column references.

**Usage Notes:**

| Required? | Data Type |
|-----------|-----------|
| No | String (column name) |

## Examples

> **Tip:** For additional examples, see *Common Tasks*.

**Example - Clean up marketing contact data with replace, set, and extract**

This example illustrates the different uses of the replacement transformations to replace or extract cell data.

**Source:**

The following dataset contains contact information that has been gathered by your marketing platform from actions taken by visitors on your website. You must clean up this data and prepare it for use in an analytics platform.

| LeadId | LastName | FirstName | Title | Phone | Request |
|--------|----------|-----------|-------|-------|---------|
| LE160301001 | Jones | Charles | Chief Technical Officer | 415-555-1212 | reg |
| LE160301002 | Lyons | Edward | | 415-012-3456 | download whitepaper |
| LE160301003 | Martin | Mary | CEO | 510-555-5555 | delete account |

| LE160301004 | Smith | Talia | Engineer | 510-123-4567 | free trial |

**Transformation:**

**Title column:** For example, you first notice that some data is missing. Your analytics platform recognizes the string value, "`#MISSING#`" as an indicator of a missing value. So, you click the missing values bar in the Title column. Then, you select the Replace suggestion card. Note that the default replacement is a null value, so you click **Edit** and update it:

| Transformation Name | Edit column with formula |
| --- | --- |
| **Parameter: Columns** | Title |
| **Parameter: Formula** | if(ismissing([Title]),'#MISSING#',Title) |

**Request column:** In the Request column, you notice that the `reg` entry should be cleaned up. Add the following transformation, which replaces that value:

| Transformation Name | Replace text or pattern |
| --- | --- |
| **Parameter: Column** | Request |
| **Parameter: Find** | `` `{start}reg{end}` `` |
| **Parameter: Replace with** | Registration |

The above transformation uses a Pattern as the expression of the `on:` parameter. This expression indicates to match from the start of the cell value, the string literal `reg`, and then the end of the cell value, which matches on complete cell values of `reg` only.

This transformation works great on the sample, but what happens if the value is `Reg` with a capital `R`? That value might not be replaced. To improve the transformation, you can modify the transformation with the following Pattern in the `on` parameter, which captures differences in capitalization:

| Transformation Name | Replace text or pattern |
| --- | --- |
| **Parameter: Column** | Request |
| **Parameter: Find** | `` `{start}{[R\|r]}eg{end}` `` |
| **Parameter: Replace with** | 'Registration' |

Add the above transformation to your recipe. Then, it occurs to you that all of the values in the `Request` column should be capitalized in title or proper case:

| Transformation Name | Edit column with formula |
| --- | --- |
| **Parameter: Columns** | Request |
| **Parameter: Formula** | proper(Request) |

Now, all values are capitalized as titles.

**Phone column:** You might have noticed some issues with the values in the `Phone` column. In the United States, the prefix `555` is only used for gathering information; these are invalid phone numbers.

In the data grid, you select the first instance of `555` in the column. However, it selects all instances of that pattern, including ones that you don't want to modify. In this case, continue your selection by selecting the similar instance of `555` in the other row. In the suggestion cards, you click the Replace Text or Pattern transformation.

Notice, however, that the default Replace Text or Pattern transformation has also highlighted the second `555` pattern in one instance, which could be a problem in other phone numbers not displayed in the sample. You must modify the selection pattern for this transformation. In the `on:` parameter below, the Pattern has been modified to match only the instances of `555` that appear in the second segment in the phone number format:

| Transformation Name | Replace text or pattern |
|---|---|
| Parameter: Column | Phone |
| Parameter: Find | `` `{start}%{3}-555-%*{end}` `` |
| Parameter: Replace with | '#INVALID#' |
| Parameter: Match all occurrences | true |

Note the wildcard construct has been added (`%*`). While it might be possible to add a pattern that matches on the last four characters exactly (`%{4}`), that matching pattern would not capture the possibility of a phone number having an extension at the end of it. The above expression does.

> **NOTE:** The above transformation creates values that are mismatched with the Phone Number data type. In this example, however, these mismatches are understood to be for the benefit of the system consuming your Trifacta output.

**LeadId column:** You might have noticed that the lead identifier column (`LeadId`) contains some embedded information: a date value and an identifier for the instance within the day. The following steps can be used to break out this information. The first one creates a separate working column with this information, which allows us to preserve the original, unmodified column:

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | LeadId |
| Parameter: New column name | 'LeadIdworking' |

You can now work off of this column to create your new ones. First, you can use the following replace transformation to remove the leading two characters, which are not required for the new columns:

| Transformation Name | Replace text or pattern |
|---|---|
| Parameter: Column | LeadIdworking |
| Parameter: Find | 'LE' |
| Parameter: Replace with | '' |

Notice that the date information is now neatly contained in the first characters of the working column. Use the following to extract these values to a new column:

| Transformation Name | Extract text or pattern |
|---|---|
| | |

| Parameter: Column to extract from | LeadIdworking |
|---|---|
| Parameter: Option | Custom text or pattern |
| Parameter: Text to extract | `` `{start}%{6}` `` |

The new `LeadIdworking2` column now contains only the date information. Cleaning up this column requires reformatting the data, retyping it as a Datetime type, and then applying the `dateformat` function to format it to your satisfaction. These steps are left as a separate exercise.

For now, let's just rename the column:

| Transformation Name | Rename columns |
|---|---|
| Parameter: Option | Manual rename |
| Parameter: Column | LeadIdworking1 |
| Parameter: New column name | 'LeadIdDate' |

In the first working column, you can now remove the date information using the following:

| Transformation Name | Replace text or pattern |
|---|---|
| Parameter: Column | LeadIdworking |
| Parameter: Find | `` `{start}%{6}` `` |
| Parameter: Replace with | '' |

You can rename this column to indicate it is a daily identifier:

| Transformation Name | Rename columns |
|---|---|
| Parameter: Option | Manual rename |
| Parameter: Column | LeadIdworking |
| Parameter: New column name | 'LeadIdDaily' |

**Results:**

| LeadId | LeadIdDaily | LeadIdDate | LastName | FirstName | Title | Phone | Request |
|---|---|---|---|---|---|---|---|
| LE160301001 | 001 | 160301 | Jones | Charles | Chief Technical Officer | #INVALID# | Registration |
| LE160301002 | 002 | 160301 | Lyons | Edward | #MISSING# | 415-012-3456 | Download Whitepaper |
| LE160301003 | 003 | 160301 | Martin | Mary | CEO | #INVALID# | Delete Account |
| LE160301004 | 004 | 160301 | Smith | Talia | Engineer | 510-123-4567 | Free Trial |

### Example - Using $col placeholder

This example illustrates how to use the conditional calculation functions.

**Functions**:

| Item | Description |
|---|---|
| AVERAGEIF Function | Generates the average value of rows in each group that meet a specific condition. Generated value is of Decimal type. |
| MINIF Function | Generates the minimum value of rows in each group that meet a specific condition. Inputs can be Integer, Decimal, or Datetime. |
| MAXIF Function | Generates the maximum value of rows in each group that meet a specific condition. Inputs can be Integer, Decimal, or Datetime. |
| VARIF Function | Generates the variance of values by group in a column that meet a specific condition. |
| STDEVIF Function | Generates the standard deviation of values by group in a column that meet a specific condition. |

**Source:**

Here is some example weather data:

| date | city | rain | temp | wind |
|---|---|---|---|---|
| 1/23/17 | Valleyville | 0.00 | 12.8 | 6.7 |
| 1/23/17 | Center Town | 0.31 | 9.4 | 5.3 |
| 1/23/17 | Magic Mountain | 0.00 | 0.0 | 7.3 |
| 1/24/17 | Valleyville | 0.25 | 17.2 | 3.3 |
| 1/24/17 | Center Town | 0.54 | 1.1 | 7.6 |
| 1/24/17 | Magic Mountain | 0.32 | 5.0 | 8.8 |
| 1/25/17 | Valleyville | 0.02 | 3.3 | 6.8 |
| 1/25/17 | Center Town | 0.83 | 3.3 | 5.1 |
| 1/25/17 | Magic Mountain | 0.59 | -1.7 | 6.4 |
| 1/26/17 | Valleyville | 1.08 | 15.0 | 4.2 |
| 1/26/17 | Center Town | 0.96 | 6.1 | 7.6 |
| 1/26/17 | Magic Mountain | 0.77 | -3.9 | 3.0 |
| 1/27/17 | Valleyville | 1.00 | 7.2 | 2.8 |
| 1/27/17 | Center Town | 1.32 | 20.0 | 0.2 |
| 1/27/17 | Magic Mountain | 0.77 | 5.6 | 5.2 |
| 1/28/17 | Valleyville | 0.12 | -6.1 | 5.1 |
| 1/28/17 | Center Town | 0.14 | 5.0 | 4.9 |
| 1/28/17 | Magic Mountain | 1.50 | 1.1 | 0.4 |
| 1/29/17 | Valleyville | 0.36 | 13.3 | 7.3 |
| 1/29/17 | Center Town | 0.75 | 6.1 | 9.0 |
| 1/29/17 | Magic Mountain | 0.60 | 3.3 | 6.0 |

**Transformation:**

The following computes average temperature for rainy days by city:

| Transformation Name | New formula |
|---|---|
| | |

| Parameter: Formula type | Single row formula |
|---|---|
| Parameter: Formula | AVERAGEIF(temp, rain > 0) |
| Parameter: Group rows by | city |
| Parameter: New column name | 'avgTempWRain' |

The following computes maximum wind for sub-zero days by city:

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | MAXIF(wind,temp < 0) |
| Parameter: Group rows by | city |
| Parameter: New column name | 'maxWindSubZero' |

This step calculates the minimum temp when the wind is less than 5 mph by city:

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | MINIF(temp,wind<5) |
| Parameter: Group rows by | city |
| Parameter: New column name | 'minTempWind5' |

This step computes the variance in temperature for rainy days by city:

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | VARIF(temp,rain >0) |
| Parameter: Group rows by | city |
| Parameter: New column name | 'varTempWRain' |

The following computes the standard deviation in rainfall for Center Town:

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | STDEVIF(rain,city=='Center Town') |
| Parameter: Group rows by | city |
| Parameter: New column name | 'stDevRainCT' |

You can use the following transforms to format the generated output. Note the `$col` placeholder value for the multi-column transforms:

| Transformation Name | Edit column with formula |
|---|---|
| Parameter: Columns | stDevRainCenterTown,maxWindSubZero |
| Parameter: Formula | numformat($col,'##.##') |

Since the following rely on data that has only one significant digit, you should format them differently:

| Transformation Name | Edit column with formula |
|---|---|
| Parameter: Columns | varTempWRain,avgTempWRain,minTempWind5 |
| Parameter: Formula | numformat($col,'##.#') |

### Results:

| date | city | rain | temp | wind | avgTempWRain | maxWindSubZero | minTempWind5 | varTempWRain | stDevRain |
|---|---|---|---|---|---|---|---|---|---|
| 1/23/17 | Valleyville | 0.00 | 12.8 | 6.7 | 8.3 | 5.1 | 7.2 | 63.8 | 0.37 |
| 1/23/17 | Center Town | 0.31 | 9.4 | 5.3 | 7.3 | | 5 | 32.6 | 0.37 |
| 1/23/17 | Magic Mountain | 0.00 | 0.0 | 7.3 | 1.6 | 6.43 | -3.9 | 12 | 0.37 |
| 1/24/17 | Valleyville | 0.25 | 17.2 | 3.3 | 8.3 | 5.1 | 7.2 | 63.8 | 0.37 |
| 1/24/17 | Center Town | 0.54 | 1.1 | 7.6 | 7.3 | | 5 | 32.6 | 0.37 |
| 1/24/17 | Magic Mountain | 0.32 | 5.0 | 8.8 | 1.6 | 6.43 | -3.9 | 12 | 0.37 |
| 1/25/17 | Valleyville | 0.02 | 3.3 | 6.8 | 8.3 | 5.1 | 7.2 | 63.8 | 0.37 |
| 1/25/17 | Center Town | 0.83 | 3.3 | 5.1 | 7.3 | | 5 | 32.6 | 0.37 |
| 1/25/17 | Magic Mountain | 0.59 | -1.7 | 6.4 | 1.6 | 6.43 | -3.9 | 12 | 0.37 |
| 1/26/17 | Valleyville | 1.08 | 15.0 | 4.2 | 8.3 | 5.1 | 7.2 | 63.8 | 0.37 |
| 1/26/17 | Center Town | 0.96 | 6.1 | 7.6 | 7.3 | | 5 | 32.6 | 0.37 |
| 1/26/17 | Magic Mountain | 0.77 | -3.9 | 3.0 | 1.6 | 6.43 | -3.9 | 12 | 0.37 |
| 1/27/17 | Valleyville | 1.00 | 7.2 | 2.8 | 8.3 | 5.1 | 7.2 | 63.8 | 0.37 |
| 1/27/17 | Center Town | 1.32 | 20.0 | 0.2 | 7.3 | | 5 | 32.6 | 0.37 |

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 1/27/17 | Magic Mountain | 0.77 | 5.6 | 5.2 | 1.6 | 6.43 | -3.9 | 12 | 0.37 |
| 1/28/17 | Valleyville | 0.12 | -6.1 | 5.1 | 8.3 | 5.1 | 7.2 | 63.8 | 0.37 |
| 1/28/17 | Center Town | 0.14 | 5.0 | 4.9 | 7.3 | | 5 | 32.6 | 0.37 |
| 1/28/17 | Magic Mountain | 1.50 | 1.1 | 0.4 | 1.6 | 6.43 | -3.9 | 12 | 0.37 |
| 1/29/17 | Valleyville | 0.36 | 13.3 | 7.3 | 8.3 | 5.1 | 7.2 | 63.8 | 0.37 |
| 1/29/17 | Center Town | 0.75 | 6.1 | 9.0 | 7.3 | | 5 | 32.6 | 0.37 |
| 1/29/17 | Magic Mountain | 0.60 | 3.3 | 6.0 | 1.6 | 6.43 | -3.9 | 12 | 0.37 |

# Settype Transform

**Contents:**

---

> **NOTE:** Transforms are a part of the underlying language, which is not directly accessible to users. This content is maintained for reference purposes only. For more information on the user-accessible equivalent to transforms, see *Transformation Reference*.

Sets the data type of the specified column or columns. The column data is validated against the new data type, which can change the results of column profiling.

Type is specified as a string literal or comma-separated set of literals. For more information on valid string literals, see *Valid Data Type Strings* .

**Tips:**

> **Tip:** You can use the `settype` transform to override the data type inferred for a column. However, if a new transformation step is added, the column data type is re-inferred, which may override your specific typing. You should consider applying `setttype` transforms as late as possible in your recipes.

- When a column is set to a data type, all values in the column are validated against the new type, which might change the number of mismatched values. Some cleanup might be required. Some operations might cause the data type to be re-validated automatically.
- It might be easier to set type using the column's drop-down. Selections of data type from the column drop-down are turned into recipe steps using the `settype` transform.
- If you encounter a significant number of mismatches after you change the data type, you might find it helpful to change or revert the type to String. All data can be interpreted as a String or a list of string values. The transforms and functions for manipulating String data might be easier to use to clean up mismatched data before changing the data type to the preferred one.
- Row values that do not match the new data type might be turned to null values during job execution.

## Basic Usage

**Single-column example:**

```
settype col: Score type: 'Integer'
```

**Output:** Changes the data type for the `Score` column to Integer.

**Multi-column example:**

```
settype col: Score,studentId type: 'Integer'
```

**Output:** Changes the data type for the `Score` and `studentId` columns to Integer.

## Syntax and Parameters

```
settype col:col1,col2 type:'string_literal'
```

| Token | Required? | Data Type | Description |
|---|---|---|---|
| settype | Y | transform | Name of the transform |
| col | Y | string | Comma-separated list of columns to which to apply the specified type. |
| type | Y | string | String literal identifying the data type to apply to the column(s). See *Valid Data Type Strings*. |

For more information on syntax standards, see *Language Documentation Syntax Notes*.

**col**

Identifies the column(s) to which to apply the transform. You can specify one or more columns.

**Usage Notes:**

| Required? | Data Type |
|---|---|
| Yes | Comma-separated strings (column name or names) |

**type**

Defines the data type that is to be applied to the transform. Type is defined as a String literal.

> **NOTE:** When specifying a data type by name, you must use the String value listed below. The Data Type value is the display name for the type.

For a list of valid strings, see *Valid Data Type Strings*.

```
settype col: zips type:'Zipcode'
```

**Output:** Changes the data type of the `zips` column to Zip Code data type. All values are validated as U.S. Zip code.

**Usage Notes:**

| Required? | Data Type |
|---|---|
| Yes | String value |

## Examples

> **Tip:** For additional examples, see *Common Tasks*.

**Example - Simple settype with date values**

**Source:**

Here is a list of activities listed by date. Note the variation in date values, including what is clearly an invalid date. Here is the source data:

```
myDate, myAction
4/4/2016,Woke up at 6:30
4-4-2016,Got ready
9-9-9999,Drove kids to school
4-4-2016, Commuted to work
```

**Transformation:**

When this data is imported into the Transformer page, there are couple of immediate issues: no column headings and blank rows at the bottom. These two transformations fix that:

| | |
|---|---|
| **Transformation Name** | Rename column with row(s) |
| **Parameter: Option** | Use row(s) as column names |
| **Parameter: Type** | Use a single row to name columns |
| **Parameter: Row number** | 1 |

| | |
|---|---|
| **Transformation Name** | Filter rows |
| **Parameter: Condition** | Custom formula |
| **Parameter: Type of formula** | Custom single |
| **Parameter: Condition** | ismissing([myDate]) |
| **Parameter: Action** | Delete matching rows |

For the invalid date, you can infer from the rows around it that it should be from the same date. You can make the following change to fix it:

| | |
|---|---|
| **Transformation Name** | Replace text or pattern |
| **Parameter: Column** | myDate |
| **Parameter: Find** | `9-9-9999` |
| **Parameter: Replace with** | '4-4-2016' |
| **Parameter: Match all occurrences** | true |

Now that the dates look fairly consistent, you can set the data type of the column to a matching Datetime format:

| | |
|---|---|
| **Transformation Name** | Change column data type |
| **Parameter: Columns** | myDate |
| **Parameter: New type** | Custom or Date/Time |
| **Parameter: Specify type** | 'mm-dd-yy','mm*dd*yyyy' |

Note the syntax above for specifying Datetime types. In addition to the `Datetime` keyword, you must specify the format type, followed by the variation of that format.

> **Tip:** A set of supported formats and variations for Datetime are available through the column data type selector. When you select your desired Datetime format, the `setttype` transform is added to your recipe.

**Results:**

| myDate | myAction |
|--------|----------|
| 4/4/2016 | Woke up at 6:30 |
| 4-4-2016 | Got ready |
| 4-4-2016 | Drove kids to school |
| 4-4-2016 | Commuted to work |

## Example - Use merge and settype to clean up numeric data that should be treated as other data types

This example illustrates how to clean up data by changing its data type to String, manipulating it using String functions, and then retyping the data to its proper data type.

**Functions:**

| Item | Description |
|------|-------------|
| IF Function | The `IF` function allows you to build if/then/else conditional logic within your transforms. |
| LEN Function | Returns the number of characters in a specified string. String value can be a column reference or string literal. |
| MERGE Function | Merges two or more columns of String type to generate output of String type. Optionally, you can insert a delimiter between the merged values. |

**Source:**

The following example contains customer ID and Zip code information in two columns. When this data is loaded into the Transformer page, it is initially interpreted as numeric, since it contains all numerals.

The four-digit `ZipCode` values should have five digits, with a `0` in front.

| CustId | ZipCode |
|--------|---------|
| 4020123 | 1234 |
| 2012121 | 94105 |
| 3212012 | 94101 |
| 1301212 | 2020 |

**Transformation:**

**CustId column:** This column needs to be retyped as String values. You can set the column data type to String through the column drop-down, which is rendered as the following transformation:

| | |
|---|---|
| **Transformation Name** | `Change column data type` |
| **Parameter: Columns** | `CustId` |

| Parameter: New type | String |
|---|---|

While the column is now of String type, future transformations might cause it to be re-inferred as Integer values. To protect against this possibility, you might want to add a marker at the front of the string. This marker should be removed prior to execution.

The basic method is to create a new column containing the customer ID marker (C) and then merge this column and the existing CustId column together. It's useful to add such an indicator to the front in case the customer identifier is a numeric value that could be confused with other numeric values. Also, this merge step forces the value to be interpreted as a String value, which is more appropriate for an identifier.

| Transformation Name | Merge columns |
|---|---|
| Parameter: Columns | 'C',CustId |

You can now delete the CustId columns and rename the new column as CustId.

**ZipCode column:** This column needs to be converted to valid Zip Code values. For ease of use, this column should be of type String:

| Transformation Name | Change column data type |
|---|---|
| Parameter: Columns | ZipCode |
| Parameter: New type | Zipcode |

The transformation below changes the value in the ZipCode column if the length of the value is four in any row. The new value is the original value prepended with the numeral 0:

| Transformation Name | Edit column with formula |
|---|---|
| Parameter: Columns | ZipCode |
| Parameter: Formula | if(len($col) == 4, merge(['0',$col]), $col) |

This column might now be re-typed as Zipcode type.

**Results:**

| CustId | ZipCode |
|---|---|
| C4020123 | 01234 |
| C2012121 | 94105 |
| C3212012 | 94101 |
| C1301212 | 02020 |

Remember to remove the C marker from the CustId column. Select the C value in the CustId column and choose the replace transform. You might need to re-type the cleaned data as String data.

# Sort Transform

**Contents:**

---

> **NOTE:** Transforms are a part of the underlying language, which is not directly accessible to users. This content is maintained for reference purposes only. For more information on the user-accessible equivalent to transforms, see *Transformation Reference.*

Sorts the dataset based on one or more columns in ascending or descending order. You can also sort based on the order of rows when the dataset was created.

**Limitations:**

> **NOTE:** This transform is intended primarily for use in the Transformer page. Sort order may not be preserved in the output files.

- If you generate a new sample after a `sort` transform has been applied, the sort order is not retained. You can re-apply the sort step, although the following limitations still apply.
- Sort order is not preserved on output when the output is a multi-part file.

## Basic Usage

```
sort order:LastName
```

**Output:** Dataset is sorted in alphabetically ascending order based on the values in the `LastName` column, assuming that the values are strings.

## Syntax and Parameters

```
sort order:column_ref
```

| Token | Required? | Data Type | Description |
|-------|-----------|-----------|-------------|
| sort | Y | transform | Name of the transform |
| order | Y | string | Name of column or columns by which to sort |

For more information on syntax standards, see *Language Documentation Syntax Notes.*

**order**

Identifies the column or set of columns by which the dataset is sorted.

- Multiple column names can be separated by commas.
- Ranges of columns cannot be specified.

The order can be reversed by adding a negative sign in front of the column name:

| Transformation Name | `Sort rows` |
|---|---|
| **Parameter: Sort by** | `-ProductName` |

**Multi-column sorts:** You can also specify multi-column sorts. The following example sorts first by the inverse order of `ProductName` and within that sort, rows are sorted by `ProductColor`:

| Transformation Name | `Sort rows` |
|---|---|
| **Parameter: Sort by** | `-ProductName,ProductColor` |

**Sort by original row numbers:** As an input value, this parameter also accepts the `SOURCEROWNUMBER` function, which performs the sort according to the original order of rows when the dataset was created.

| Transformation Name | `Sort rows` |
|---|---|
| **Parameter: Sort by** | `$sourcerownumber` |

See *SOURCEROWNUMBER Function*.

**Usage Notes:**

| Required? | Data Type |
|---|---|
| Yes | String (column name) |

Data is sorted based on the data type of the source:

| Data Type of Source | Sort Order |
|---|---|
| Integer | Numerical |
| Decimal | Numerical |
| Datetime | Numerical |
| All others | String |

## Examples

> **Tip:** For additional examples, see *Common Tasks*.

**Example - sort methods**

**Source:**

The column without a name identifies the original row numbers. In the data grid, this information is available when you hover over the black dot to the left of a row of data.

| | CustId | FirstName | LastName | City | State | LastOrder |
|---|---|---|---|---|---|---|
| 1 | 1001 | Skip | Jones | San Francisco | CA | 25 |
| 2 | 1002 | Adam | Allen | Oakland | CA | 1099 |

| 3 | 1003 | David | Wiggins | Oakland | MI | 125.25 |
| 4 | 1004 | Amanda | Green | Detroit | MI | 452.5 |
| 5 | 1005 | Colonel | Mustard | Los Angeles | CA | 950 |
| 6 | 1006 | Pauline | Hall | Saginaw | MI | 432.22 |
| 7 | 1007 | Sarah | Miller | Cheyenne | WY | 724.22 |
| 8 | 1008 | Teddy | Smith | Juneau | AK | 852.11 |
| 9 | 1009 | Joelle | Higgins | Sacramento | CA | 100 |

**Transformation:**

First, you might want to clean up the number formatting in the `lastOrder` column. The following formats the values to always include two digits after the decimal point:

| Transformation Name | Edit column with formula |
|---|---|
| Parameter: Columns | LastOrder |
| Parameter: Formula | numformat(LastOrder, '####.00') |

Now, you're interested in the highest value for your customers' most recent orders. You can apply the following sort:

| Transformation Name | Sort rows |
|---|---|
| Parameter: Sort by | -LastOrder |

Rows are sorted by the `LastOrder` column in descending order (largest to smallest):

|   | CustId | FirstName | LastName | City | State | LastOrder |
|---|---|---|---|---|---|---|
| 2 | 1002 | Adam | Allen | Oakland | CA | 1099.00 |
| 5 | 1005 | Colonel | Mustard | Los Angeles | CA | 950.00 |
| 8 | 1008 | Teddy | Smith | Juneau | AK | 852.11 |
| 7 | 1007 | Sarah | Miller | Cheyenne | WY | 724.22 |
| 4 | 1004 | Amanda | Green | Detroit | MI | 452.50 |
| 6 | 1006 | Pauline | Hall | Saginaw | MI | 432.22 |
| 3 | 1003 | David | Wiggins | Oakland | MI | 125.25 |
| 9 | 1009 | Joelle | Higgins | Sacramento | CA | 100.00 |
| 1 | 1001 | Skip | Jones | San Francisco | CA | 25.00 |

The above row numbers represent the original order of the rows. Now, you want to get your data geographically organized by sorting by city and state. You can perform multi-column sorts such as the following, which sorts first by `State` and then by `City` columns:

| Transformation Name | Sort rows |
|---|---|
| Parameter: Sort by | State,City |

In the generated output, the data is first sorted by the `State` value. Each set of rows within the same `State` value is also sorted by the `City` value.

|   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|

|   | CustId | FirstName | LastName | City | State | LastOrder |
|---|--------|-----------|----------|------|-------|-----------|
| 8 | 1008 | Teddy | Smith | Juneau | AK | 852.11 |
| 5 | 1005 | Colonel | Mustard | Los Angeles | CA | 950.00 |
| 2 | 1002 | Adam | Allen | Oakland | CA | 1099.00 |
| 9 | 1009 | Joelle | Higgins | Sacramento | CA | 100.00 |
| 1 | 1001 | Skip | Jones | San Francisco | CA | 25.00 |
| 4 | 1004 | Amanda | Green | Detroit | MI | 452.50 |
| 3 | 1003 | David | Wiggins | Oakland | MI | 125.25 |
| 6 | 1006 | Pauline | Hall | Saginaw | MI | 432.22 |
| 7 | 1007 | Sarah | Miller | Cheyenne | WY | 724.22 |

### Example - Sort by original row numbers

This example illustrates how you can rename columns based on the contents of specified rows.

**Source:**

You have imported the following racer data on heat times from a CSV file. When loaded in the Transformer page, it looks like the following:

| (rowId) | column2 | column3 | column4 | column5 |
|---------|---------|---------|---------|---------|
| 1 | Racer | Heat 1 | Heat 2 | Heat 3 |
| 2 | Racer X | 37.22 | 38.22 | 37.61 |
| 3 | Racer Y | 41.33 | DQ | 38.04 |
| 4 | Racer Z | 39.27 | 39.04 | 38.85 |

In the above, the `(rowId)` column references the row numbers displayed in the data grid; it is not part of the dataset. This information is available when you hover over the black dot on the left side of the screen.

**Transformation:**

You have examined the best performance in each heat according to the sample. You then notice that the data contains headers, but you forget how it was originally sorted. The data now looks like the following:

| (rowId) | column2 | column3 | column4 | column5 |
|---------|---------|---------|---------|---------|
| 1 | Racer Y | 41.33 | DQ | 38.04 |
| 2 | Racer | Heat 1 | Heat 2 | Heat 3 |
| 3 | Racer X | 37.22 | 38.22 | 37.61 |
| 4 | Racer Z | 39.27 | 39.04 | 38.85 |

You can use the following transformation to use the third row as your header for each column:

| | |
|---|---|
| **Transformation Name** | `Rename column with row(s)` |
| **Parameter: Option** | `Use row(s) as column names` |
| **Parameter: Type** | `Use a single row to name columns` |
| **Parameter: Row number** | 3 |

**Results:**

After you have applied the above transformation, your data should look like the following:

| (rowId) | Racer | Heat_1 | Heat_2 | Heat_3 |
|---|---|---|---|---|
| 3 | Racer Y | 41.33 | DQ | 38.04 |
| 2 | Racer X | 37.22 | 38.22 | 37.61 |
| 4 | Racer Z | 39.27 | 39.04 | 38.85 |

# Split Transform

**Contents:**

---

> **NOTE:** Transforms are a part of the underlying language, which is not directly accessible to users. This content is maintained for reference purposes only. For more information on the user-accessible equivalent to transforms, see *Transformation Reference.*

Splits the specified column into separate columns of data based on the delimiters in the transform. Delimiters can be specified in a number of literal or pattern-based methods. Whitespace delimiters are supported.

This transform might be automatically applied as one of the first steps of your recipe.

> **Tip:** When this transform appears in a suggestion card, the maximum number of suggested columns to split is 250, which may prevent the browser from crashing. If your dataset requires additional column splits, you can edit the transformation and increase the maximum number of splits. Avoid creating datasets that are wider than 1000 columns.

When the `split` transform is applied, the source column is dropped.

- Before applying this transform, you can create a copy of the source column using the `derive` transform. See *Derive Transform*.
- To retain the source column, you can use the `extract` transform and pattern-based matching. See *Extract Transform*.

## Basic Usage

```
split col: MyValues on: ',' limit: 3
```

**Output:** Splits the source `MyValues` column into four separate columns. Values in the columns are determined based on the comma (`,`) delimiter. If a row only has two commas in it, then the final generated column is null.

## Syntax and Parameters

```
split col:column_ref [quote:'quoted_string'] [ignoreCase:true|false] [limit:int_num]
[after:start_point | from: start_point] [before:end_point | to:end_point]
[on:'exact_match'] [at:(start_index,end_index)] [delimiters:'string1','string2',
'string3'] [positions: int1,int2,int3] [every:int_num]
```

For more information on syntax standards, see *Language Documentation Syntax Notes*.

The `split` transform supports the following general methods for specifying the delimiters by which to split the column. Depending on your use of the transform, different sets of parameters apply.

| Delimiter Methods | Description | Operative Parameters |
|---|---|---|
| single-pattern delimiters | Column is split based on one of the following: <br><br> 1) patterns used to describe the beginning and ending of the field delimiter(s), <br><br> 2) single delimiter, which may be repeated, <br><br> 3) index values of the start and end points of the delimiter | `quote` <br><br> At least one of the following parameters must be specified: <br><br> `after`, `at`, `from`, `before`, `on`, `to` |
| multi-pattern delimiters | Column is split based one of the following literal methods: <br><br> 1) explicit sequence of delimiters, <br><br> 2) explicit list of character index positions <br><br> 3) every *N* characters | At least one of the following parameters must be specified: <br><br> `delimiters`, `positions`, `every` |

**Shared parameters:**

The following parameters are shared between the operating modes:

| Token | Required? | Data Type | Description |
|---|---|---|---|
| split | Y | transform | Name of the transform |
| col | Y | string | Source column name |
| ignoreCase | N | boolean | If `true`, matching is case-insensitive. |
| limit | N | integer (positive) | Specifies the maximum of columns to split from the source column |

**Single-pattern delimiter parameters:**

> **Tip:** For this method of matching, at least one of the following parameters must be used: `at`, `before`, `from`, `on`, or `to`.

| Token | Required? | Data Type | Description |
|---|---|---|---|
| after | N | string | String literal or pattern that precedes the pattern to match |
| at | N | Array | Two-integer array identifying the character indexes of start and end characters to match |
| before | N | string | String literal or pattern that appears after the pattern to match |
| from | N | string | String literal or pattern that identifies the start of the pattern to match |
| on | N | string | String literal or pattern that identifies the pattern to match. |
| to | N | string | String literal or pattern that identifies the end of the pattern to match |
| quote | N | string | Specifies a quoted object that is omitted from pattern matching |

**Multi-pattern delimiter parameters:**

> **Tip:** Use one of the following parameters for this method of matching. Do not use combinations of them.

| Token | Required? | Data Type | Description |
|---|---|---|---|
| delimiters | N | array | Array of strings that list the explicit field delimiters in the order to apply them to the column. |
| positions | N | array | Array of integers that identify the zero-based character index values where to split the column. |
| every | N | integer | String literal or pattern that appears after the pattern to match |

### col

Identifies the column to which to apply the transform. You can specify only one column.

```
split col: MyCol on: 'MyString'
```

**Output:** Splits the `MyCol` column into two separate columns whose values are to the left and right of the `MyStri ng` value in each cell.

- If a delimiter value is not detected, the cell value appears in the first of the new columns.
- When the `limit` parameter is not specified, the default value of `1` is applied.

**Usage Notes:**

| Required? | Data Type |
|---|---|
| Yes | String (column name) |

### ignoreCase

Indicates whether the match should ignore case or not.

- Set to `true` to ignore case matching.
- (Default) Set to `false` to perform case-sensitive matching.

```
split col: MyCol on: 'My String' ignoreCase: true
```

**Output:** Splits the `MyCol` column on case-insensitive versions of the `on` parameter value, if they appear in cell values: `My String`, `my string`, `My string`, etc.

**Usage Notes:**

| Required? | Data Type |
|---|---|
| No | Boolean |

### limit

The `limit` parameter defines the maximum number of times that a pattern can be matched within a column.

> **NOTE:** The `limit` parameter cannot be used with the following parameters: `at`, `positions`, or `delimi ters`.

A set of new columns is generated, as defined by the `limit` parameter. Each matched instance populates a separate column, until there are no more matches or all of the `limit`-generated new columns are filled.

```
split col: MyCol on: 'z' limit: 3
```

**Output:** Splits the `MyCol` column on each instance of the letter `z`, generating 4 new columns. If there are fewer than 3 instances of `z` in a cell, the corresponding columns after the split are blank.

> **NOTE:** Avoid creating datasets that are wider than 2500 columns. Performance can degrade significantly on very wide datasets.

**Usage Notes:**

| Required? | Data Type |
|-----------|-----------|
| No | Integer (positive) |

- Defines the number of columns that can be created by the `split` transform.
- If not specified, exactly one column is created. See *Pattern Clause Position Matching*.

**after**

A pattern identifier that precedes the value or pattern to match. Define the `after` parameter value using string literals, regular expressions, or Patterns .

> If this parameter is the only pattern describer:
> - The column is split into two. The first column contains the part of the source column before the `after` matching value. The second column is blank.
> - If the value appears more than once, no additional splitting is made, since there is no other pattern parameter.
>
> > **NOTE:** For `after`, `before`, `from`, and `to`, matching occurs only one time at most. Additional instances of the parameter's value in the cell do not cause another column split. For more predictable results, you should specify another pattern parameter.

- If the `after` value does not appear in the column, the original column value is written to the first split column.
- This parameter is typically used with another to describe a field delimiting pattern. See below.

```
split col: MyCol after: '\' before:'|'
```

**Output:** Splits values in `MyCol` based on value between the two characters. The first column contains the part of the `MyCol` that appears before the backslash (\), and the second column contains the part of `MyCol` that appears after the pipe character (|). The content between the delimiting characters is dropped.

**Usage Notes:**

| Required? | Data Type |
|-----------|-----------|
| No | String or pattern |

- The `after` and `from` parameters are very similar. `from` includes the matching value as part of the delimiter string.
- `after` can be used with either `to` or `before`. See *Pattern Clause Position Matching*.

## at

Identifies the start and end point of the pattern to interest.

Parameter inputs are in the form of $x,y$ where $x$ and $y$ are positive integers indicating the starting character and ending character, respectively, of the pattern of interest.

- $x$ must be less than $y$.
- If $y$ is greater than the length of the value, the pattern is defined to the end of the value, and a match is made.

```
split col: MyCol at: 2,6
```

**Output:** Splits the `MyCol` column on the value that begins at the second character in the column and extends to the sixth character of the column. Contents before the value are in the first column, and contents after the value are in the second column.

**Usage Notes:**

| Required? | Data Type |
|---|---|
| No | Array of two Integers ( `X`,`Y` ) |

The `at` parameter cannot be combined with any of the following: `on`, `after`, `before`, `from`, `to`, and `quote`. See *Pattern Clause Position Matching*.

## before

A pattern identifier that occurs after the value or pattern to match. Define the pattern using string literals, regular expressions, or Patterns .

If this parameter is the only pattern describer:
- The column is split into two. The first column is blank. The second column contains the part of the source column after the `before` matching value.
- If the value appears more than once, no additional splitting is made, since there is no other pattern parameter.

> **NOTE:** For `after`, `before`, `from`, and `to`, matching occurs only one time at most. Additional instances of the parameter's value in the cell do not cause another column split. For more predictable results, you should specify another pattern parameter.

- If the `before` value does not appear in the column, the original column value is written to the first split column.
- This parameter is typically used with another to describe a field delimiting pattern. See below.

```
split col: MyCol before: '/' from:'Go:'
```

**Output:** Splits contents of `MyCol` into two columns. The first column contains the values that appear before the `Go:` string, and the second column contains the values after the backslash.

**Usage Notes:**

| | |
|---|---|

| Required? | Data Type |
|---|---|
| No | String or pattern |

- The `before` and `to` parameters are very similar. `to` includes the matching value as part of the delimiter string.
- `before` can be used with either `from` or `after`. See *Pattern Clause Position Matching* .

### from

Identifies the pattern that marks the beginning of the value to match. It can be a string literal, Pattern , or regular expression. The `from` value is included in the match.

- If this parameter is the only pattern describer:
  - The column is split into two. The first column contains the part of the source column before the `from` matching value. The second column is blank.
  - If the value appears more than once, no additional splitting is made, since there is no other pattern parameter.

> **NOTE:** For `after`, `before`, `from`, and `to`, matching occurs only one time at most. Additional instances of the parameter's value in the cell do not cause another column split. For more predictable results, you should specify another pattern parameter.

- If the `from` value does not appear in the column, the output value is original column value.
- This parameter is typically used with another to describe a field delimiting pattern. See below.

```
split col: MyCol from: 'go:' to:'stop:'
```

**Output:** Splits contents of `MyCol` from `go:`, including `go:` to `stop:`, including `stop:`. Contents before the string appear in the first column, contents after the string appear in the second one.

**Usage Notes:**

| Required? | Data Type |
|---|---|
| No | String or pattern |

- The `after` and `from` parameters are very similar. `from` includes the matching value as part of the delimiter string.
- `from` can be used with either `to` or `before`. See *Pattern Clause Position Matching* .

### on

Identifies the pattern to match, which can be a string literal, Pattern , or regular expression.

If the value does not appear in the source column, the original value is written to the first column of the split columns.

```
split col: MyCol on: `###ERROR`
```

**Output:** Column into two columns. The first column contains values in the column appearing before `###ERROR`, and the second column contains the values appearing after this string.

> **Tip:** You can insert the Unicode equivalent character for this parameter value using a regular expression of the form `/\uHHHH/`. For example, `/\u0013/` represents Unicode character `0013` (carriage return). For more information, see *Supported Special Regular Expression Characters*.

**Usage Notes:**

| Required? | Data Type |
|---|---|
| No | String or pattern |

**to**

Identifies the pattern that marks the ending of the value to match. Pattern can be a string literal,  Patterns , or regular expression. The `to` value is included in the match.

- If this parameter is the only pattern describer:
  - The column is split into two. The first column is blank. The second column contains the part of the source column after the `to` matching value.
  - If the value appears more than once, no additional splitting is made, since there is no other pattern parameter.

  > **NOTE:** For `after`, `before`, `from`, and `to`, matching occurs only one time at most. Additional instances of the parameter's value in the cell do not cause another column split. For more predictable results, you should specify another pattern parameter.

- If the `to` value does not appear in the column, the original column value is written to the first split column.
- This parameter is typically used with another to describe a field delimiting pattern. See below.

```
split col:MyCol from:'note:' to: ` `
```

**Output:** Splits `MyCol` column all contents that appear before `note:` in the first column and all contents that appear after the first space after `note:` in the second column.

**Usage Notes:**

| Required? | Data Type |
|---|---|
| No | String or pattern |

- The `before` and `to` parameters are very similar.  `to` includes the matching value as part of the delimiter string.
- `to` can be used with either `from` or `after`. See *Pattern Clause Position Matching*.

**quote**

Can be used to specify a string as a single quoted object. This parameter value can be one or more characters.

```
split col: MyLog on: `|` limit:10 quote: '"'
```

**Output:** Splits the `MyLog` column, on the pipe character (|), while ignoring any pipe characters that are found between double-quote characters in the column. Based on the value in the `limit` parameter, the transform is limited to creating a maximum of 10 splits.

**Usage Notes:**

| Required? | Data Type |
|---|---|
| No | String |

- Parameter value is the quoted object.
- The `quote` value can appear anywhere in the column value. It is not limited by the constraints of any other parameters.

### delimiters

The `delimiters` parameter specifies a comma-separated list of string literals or patterns to identify the delimiters to use to split the data. Values can be string literals, regular expressions, or Patterns .

- The sequence of values defines the order in which the delimiters are applied.
- Values do not need to be the same.

```
split col:myCol delimiters:'|',' ',',','|'
```

**Output:** Splits the `myCol` column into four separate columns, as indicated by the sequence of delimiters.

**Usage Notes:**

> **NOTE:** Do not use the `limit` or `quote` parameters with the `delimiters` parameter.

| Required? | Data Type |
|---|---|
| No | Array of Strings (literal, regular expression, Pattern ) |

### positions

The `positions` parameter specifies a comma-separated list of integers that identify zero-based character index values at which to split the column. Values must be Integers.

```
split col:myCol positions:20,55,80
```

> **Tip:** Numeric values for positions do not need to be in sorted order.

**Output:** Splits the `myCol` column into four separate columns, where:

- `column1` = characters 0-20 from the source column,
- `column2` = characters 21-55
- `column3` = characters 56-80
- `column4` = characters 80 to the end of the cell value

**Usage Notes:**

> **NOTE:** Do not use the `limit` or `quote` parameters with the `positions` parameter.

| Required? | Data Type |
|---|---|
| No | Array of Integers (literal, regular expression, Pattern ) |

**every**

The `every` parameter can be used to specify fixed-width splitting of the source column. This Integer value defines the number of characters in each column of the split output.

If needed, you can use the `every` parameter with the `limit` parameter to define the maximum number of output columns:

```
split col:myCol every:20 limit:5
```

**Output:** Splits the `myCol` column every 20 characters, with a limit of five splits. The sixth column contains all characters after the 100th character in the cell value.

**Usage Notes:**

| Required? | Data Type |
|-----------|-----------|
| No        | Integer   |

## Pattern Groups

When you build or edit a `split` transform step in the Transform Builder, you can select one of the following pattern groups to apply to your transform. A **pattern group** is a set of related patterns that define a method of matching in a cell's data. Some pattern groups apply to multiple transforms, and some apply to the `split` transform only.

## Examples

> **Tip:** For additional examples, see *Common Tasks*.

**Example - Split with single pattern delimiters**

**Source:**

| ColA                | ColB        | ColC    |
|---------------------|-------------|---------|
| This my String That | abXcdXefXgh | 01AA001 |
| my string This That | ijXklXmnXop | 02BB002 |
| This That My String | qrXstXuvXwy | 03CC003 |

**Transformation:**

**ColA:** You can use the following transformation to split on the variations of `My String`: In this case, the `ignore Case` parameter ensures that all variations on capitalization are matched:

| Transformation Name    | Split column |
|------------------------|--------------|
| Parameter: Column      | ColA         |
| Parameter: Option      | On pattern   |
| Parameter: Match pattern | 'My String' |

| | |
|---|---|
| Parameter: Ignore case | `true` |

**ColB:** For this column, the letter `x` is the split marker, and the data is consistently formatted with three instances per row:

| | |
|---|---|
| Transformation Name | `Split column` |
| Parameter: Column | `ColB` |
| Parameter: Option | `On pattern` |
| Parameter: Match pattern | `'X'` |
| Parameter: Number of matches | 3 |

ColC: In this column, the double-letter marker varies between the rows. However, it is consistently in the same location in each row:

| | |
|---|---|
| Transformation Name | `Split column` |
| Parameter: Column | `ColC` |
| Parameter: Option | `Sequence of positions` |
| Parameter: Positions | `2,4` |

### Results:

When the above transforms are added, the source columns are dropped, leaving the following columns:

| ColA1 | ColA2 | ColB1 | ColB2 | ColB3 | ColB4 | ColC1 | ColC2 |
|---|---|---|---|---|---|---|---|
| This | That | ab | cd | ef | gh | 01 | 001 |
| | This That | ij | kl | mn | op | 02 | 002 |
| This That | | qr | st | uv | wy | 03 | 003 |

### Example - Split with quoted values

This example demonstrates how to use `quote` parameter for more sophisticated splitting of columns of data using the `split` transform.

### Source:

In this example, the following CSV data, which contains contact information, is imported into the application:

```
LastName,FirstName,Role,Company,Address,Status
Wagner,Melody,VP of Engineering,Example.com,"123 Main Street, Oakland, CA 94601",Prospect
Gruber,Hans,"Director, IT",Example.com,"456 Broadway, Burlingame, CA, 94401",Customer
Franks,Mandy,"Sr. Manager, Analytics",Tricorp,"789 Market Street, San Francisco, CA, 94105",Customer
```

### Transformationn:

When this data is pulled into the application, some initial parsing is performed for you:

| column2 | column3 | column4 | column5 | column6 | column7 |
|---|---|---|---|---|---|
| LastName | FirstName | Role | Company | Address | Status |
| Wagner | Melody | VP of Engineering | Example.com | "123 Main Street, Oakland, CA 94601" | Prospect |

| Gruber | Hans | "Director, IT" | Example.com | "456 Broadway, Burlingame, CA, 94401" | Customer |
|--------|------|----------------|-------------|----------------------------------------|----------|
| Franks | Mandy | "Sr. Manager, Analytics" | Tricorp | "789 Market Street, San Francisco, CA, 94105" | Customer |

When you open the Recipe Panel, you should see the following transforms:

| | |
|---|---|
| **Transformation Name** | `Split into rows` |
| **Parameter: Column** | `column1` |
| **Parameter: Split on** | `\n` |
| **Parameter: Ignore matches between** | `\"` |
| **Parameter: Quote escape character** | `\"` |

| | |
|---|---|
| **Transformation Name** | `Split column` |
| **Parameter: Column** | `column1` |
| **Parameter: Option** | `On pattern` |
| **Parameter: Match pattern** | `','` |
| **Parameter: Number of Matches** | `5` |
| **Parameter: Ignore matches between** | `\"` |

The first transform splits the raw source data into separate rows in the carriage return character ($\r$), ignoring all values between the double-quote characters. Note that this value must be escaped. The double-quote character does not require escaping. While there are no carriage returns within the actual data, the application recognizes that these double-quotes are identifying single values and adds the quote value.

The second transform splits each row of data into separate columns. Since it is comma-separated data, the application recognizes that this value is the column delimiter, so the `on` value is set to the comma character (`,`). In this case, the quoting is necessary, as there are commas in the values in `column4` and `column6`, which are easy to clean up.

To finish clean up of the dataset, you can promote the first row to be your column headers:

| | |
|---|---|
| **Transformation Name** | `Rename column with row(s)` |
| **Parameter: Option** | `Use row(s) as column names` |
| **Parameter: Type** | `Use a single row to name columns` |
| **Parameter: Row number** | `1` |

You can remove the quotes now. Note that the following applies to two columns:

| | |
|---|---|
| **Transformation Name** | `Replace text or patterns` |
| **Parameter: Column** | `Address,Role` |
| **Parameter: Find** | `'\"'` |
| **Parameter: Replace** | `''` |

| | |
|---|---|
| **Parameter: Match all occurrences** | `true` |

Now, you can split up the `Address` column. You can highlight one of the commas and the space after it in the column, but make sure that your final statement looks like the following:

| | |
|---|---|
| **Transformation Name** | `Split column` |
| **Parameter: Column** | `column1` |
| **Parameter: Option** | `On pattern` |
| **Parameter: Match pattern** | `','` |
| **Parameter: Number of Matches** | `2` |

Notice that there is some dirtiness to the resulting `Address3` column:

| Address3 |
|---|
| CA 94601 |
| CA, 94401 |
| CA, 94105 |

Use the following to remove the comma. In this case, it's important to leave the space between the two values in the column, so the `on` value should only be a comma. Below, the `width` value is two single quotes:

| | |
|---|---|
| **Transformation Name** | `Replace text or patterns` |
| **Parameter: Column** | `Address3` |
| **Parameter: Find** | `','` |
| **Parameter: Replace** | `''` |
| **Parameter: Match all occurrences** | `true` |

You can now split the `Address3` column on the space delimiter:

| | |
|---|---|
| **Transformation Name** | `Split by delimiter` |
| **Parameter: Column** | `Address3` |
| **Parameter: Option** | `by delimiter` |
| **Parameter: Delimiter** | `' '` |
| **Parameter: Number of columns to create** | `2` |

**Results:**

After you rename the columns, you should see the following:

| LastName | FirstName | Role | Company | Address | City | State | Zipcode | Status |
|---|---|---|---|---|---|---|---|---|
| Wagner | Melody | VP of Engineering | Example.com | 123 Main Street | Oakland | CA | 94601 | Prospect |
| Gruber | Hans | Director, IT | Example.com | 456 Broadway | Burlingame | CA | 94401 | Customer |

| Franks | Mandy | Sr. Manager, Analytics | Tricorp | 789 Market Street | San Francisco | CA | 94105 | Customer |
|--------|-------|------------------------|---------|-------------------|---------------|-----|--------|----------|

### Example - Splitting with different delimiter types

This example shows how you can split data from a single column into multiple columns using delimiters.

- **single-pattern delimiter:** One pattern is applied one or more times to the source column to define the delimiters for the output columns
- **multi-pattern delimiter:** Multiple patterns, in the form of explicit strings, character index positions, or fixed-width fields, are used to split the column.

#### Source:

In this example, your CSV dataset contains status messages from a set of servers. In this case, the data about the server and the timestamp is contained in a single value within the CSV.

```
Server|Date Time,Status
admin.examplecom|2016-03-05 07:04:00,down
webapp.examplecom|2016-03-05 07:04:00,ok
admin.examplecom|2016-03-05 07:04:30,rebooting
webapp.examplecom|2016-03-05 07:04:00,ok
admin.examplecom|2016-03-05 07:05:00,ok
webapp.examplecom|2016-03-05 07:05:00,ok
```

#### Transformation:

When the data is first loaded into the Transformer page, the CSV data is split using the following two transformations:

| Transformation Name | `Split into rows` |
|---------------------|-------------------|
| **Parameter: Column** | `column1` |
| **Parameter: Split on** | `\n` |

| Transformation Name | `Split column` |
|---------------------|----------------|
| **Parameter: Column** | `column1` |
| **Parameter: Option** | `On pattern` |
| **Parameter: Match pattern** | `','` |
| **Parameter: Ignore matches between** | `\"` |

You might need to add a `header` as the first step:

| Transformation Name | `Rename column with row(s)` |
|---------------------|-----------------------------|
| **Parameter: Option** | `Use row(s) as column names` |
| **Parameter: Type** | `Use a single row to name columns` |
| **Parameter: Row number** | `1` |

At this point, your data should look like the following:

| Server_Date_Time | Status |
|------------------|--------|

| | |
|---|---|
| admin.example.com\|2016-03-05 07:04:00 | down |
| webapp.example.com\|2016-03-05 07:04:00 | ok |
| admin.example.com\|2016-03-05 07:04:30 | rebooting |
| webapp.example.com\|2016-03-05 07:04:30 | ok |
| admin.example.com\|2016-03-05 07:05:00 | ok |
| webapp.example.com\|2016-03-05 07:05:00 | ok |

The first column contains three distinct sets of data: the server name, the date, and the time. Note that the delimiters between these fields are different, so you should use a multi-pattern delimiter to break them apart:

| | |
|---|---|
| **Transformation Name** | `Split column` |
| **Parameter: Column** | `Server\|Date Time` |
| **Parameter: Option** | `Sequence of patterns` |
| **Parameter: Pattern1** | `','` |
| **Parameter: Pattern2** | `' '` |

When the above is added, you should see three separate columns with the individual fields of information. Note that the source column has been automatically dropped.

Now, you decide that it would be useful to break apart the date information column into separate columns for year, month, and day. Since the column delimiter of this field is consistently a dash (-), you can use a single-pattern delimiter with the following transformation:

| | |
|---|---|
| **Transformation Name** | `Split by delimiter` |
| **Parameter: Column** | `Server\|Date Time2` |
| **Parameter: Option** | `By delimiter` |
| **Parameter: Delimiter** | `'-'` |
| **Parameter: Number of columns to create** | 2 |

**Results:**

After you rename the generated columns, your dataset should look like the following. Note that the source timestamp column has been automatically dropped.

| server | year | month | day | time | Status |
|---|---|---|---|---|---|
| admin.example.com | 2016 | 03 | 05 | 07:04:00 | down |
| webapp.example.com | 2016 | 03 | 05 | 07:04:00 | ok |
| admin.example.com | 2016 | 03 | 05 | 07:04:30 | rebooting |
| webapp.example.com | 2016 | 03 | 05 | 07:04:30 | ok |
| admin.example.com | 2016 | 03 | 05 | 07:05:00 | ok |
| webapp.example.com | 2016 | 03 | 05 | 07:05:00 | ok |

# Splitrows Transform

**Contents:**

---

> **NOTE:** Transforms are a part of the underlying language, which is not directly accessible to users. This content is maintained for reference purposes only. For more information on the user-accessible equivalent to transforms, see *Transformation Reference*.

Splits a column of values into separate rows of data based on the specified delimiter. You can split rows only on String literal values. Pattern-based row splitting is not supported.

> **NOTE:** The `splitrows` transform must be the first one in your recipe. When a dataset is loaded for the first time in the Transformer page, a `splitrows` transform may added as the first step of the recipe. You cannot cannot add another `splitrows` transform later in your recipe.

## Basic Usage

If you load CSV data into the Transformer page and then review the first recipe step in the Recipe panel, it might look like the following:

```
splitrows col: column1 on: '\r'
```

**Output:** The above splits all of the CSV data, which is stored as a comma-separated values in `column1` initially. The delimiter for the end of the row is a carriage return, which is indicated by the `\r` escaped value.

## Syntax and Parameters

```
splitrows col:column_ref on:'string_literal' [quote:'quoted_string']
```

| Token | Required? | Data Type | Description |
|---|---|---|---|
| splitrows | Y | transform | Name of the transform |
| col | Y | string | Source column name |
| on | Y | string | Specifies the end of row delimiter for each value in the source column |
| quote | N | string | Specifies a quoted object that is omitted from pattern matching |
| quoteEscapeChar | N | string | Specifies the escape character that is used to precede quote marks. |

For more information on syntax standards, see *Language Documentation Syntax Notes*.

**col**

Identifies the column to which to apply the transform. You can specify only one column.

**Usage Notes:**

| Required? | Data Type |
|---|---|
| Yes | String (column name) |

**on**

Identifies the pattern to match, which can be a string literal, Pattern , or regular expression.

> **NOTE:** Value must be a string. For this transform, the parameter defines the string on which to split the current row and add the data after the string to the new row.

**Usage Notes:**

| Required? | Data Type |
|---|---|
| Yes | String literal |

**quote**

Can be used to specify a string as a single quoted object.

> **NOTE:** This parameter value must be a single character.

```
splitrows col: MyCol on: '\r\n' quote: '"'
```

**Output:** Splits the `MyCol` column into separate rows on the return-newline character string (`\r\n`). Values contained within double quotes (`"`) are treated as strings, even if they contain `\r\n` values.

**Usage Notes:**

| Required? | Data Type |
|---|---|
| No | String |

**quoteEscapeChar**

By default, the platform assumes the following characters are used to escape quote marks in text-based formats that use quotes to identify fields:

- **JSON:** Platform assumes that `\` is used.
- **All other file formats:** Platform assumes that `"` is used.

Optionally, you can specify the character that is used to escape quote marks in each recipe. Typically, this value is specified for processing JSON data or for customizing the transform for your specific data.

```
splitrows col: MyCol on: '\r\n' quote: '"' quoteEscapeChar:'"'
```

**Usage Notes:**

| Required? | Data Type |
|-----------|-----------|
| No | String literal (single character) |

## Examples

> **Tip:** For additional examples, see *Common Tasks.*

**Example - splitrows with CSV data**

**Unstructured source:**

Before you import, your data in CSV format looks like the following:

```
Date,UserId,Message
3/14/16,jjones,"Hi, everyone!

Happy, St. Patrick's Day!"
3/14/16,lsmith,"@jjones, it's on 3/17."
3/14/16,thughes,lol
3/14/16,jjones,"@lsmith, no harm in celebrating twice!"
```

Notes:

- The Message value for the first row of data contains carriage returns, which must be captured in the data value and not used to split the row.
- The Message value for `thughes` is a single unquoted value.

**Transformation:**

When the data is first loaded into the Transformer page, the following step is added as the first step to the recipe:

| Transformation Name | `Split into rows` |
|---------------------|-------------------|
| **Parameter: Column** | `column1` |
| **Parameter: Split on** | `'\r'` |
| **Parameter: Quote escape character** | `'\"'` |

This transformation splits the unstructured CSV data on the carriage return. However, values that are stored between double quotes are treated as single strings, and no row breaks are applied to this data.

**Results:**

For CSV data, this step, a `split` step, and a `header` step are typically added automatically as the first steps of the recipe. In the Transformer page, this dataset should look like the following:

| Date | UserId | Message |
|------|--------|---------|
| 3/14/16 | jjones | Hi, everyone! <sup>C</sup><sub>R</sub> <sup>C</sup><sub>R</sub>Happy, St. Patrick's Day! |
| 3/14/16 | lsmith | @jjones, it's on 3/17. |
|  |  |  |

| | | |
|---|---|---|
| 3/14/16 | thughes | lol |
| 3/14/16 | jjones | @lsmith, no harm in celebrating twice! |

The $C_R$ marker is used to indicate a carriage return in the data.

# Unnest Transform

**Contents:**

---

> **NOTE:** Transforms are a part of the underlying language, which is not directly accessible to users. This content is maintained for reference purposes only. For more information on the user-accessible equivalent to transforms, see *Transformation Reference*.

Unpacks nested data from an Array or Object column to create new rows or columns based on the keys in the source data.

This transform works differently on columns of Array or Object type.

The `unnest` transform must include keys that you specify as part of the transform step. To unnest a column of array data that contains no keys, use the `flatten` transform. See *Flatten Transform*.

This transform might be automatically applied as one of the first steps of your recipe.

## Basic Usage

```
unnest col: myObj keys:'sourceA','sourceB' pluck:true markLineage:true
```

**Output:**

- Extracts from the `myObj` column the corresponding values for the keys `sourceA` and `sourceB` into two new columns.
- Since `markLineage` is `true`, these new column names are prepended with the source name: `sourceA_column1` and `sourceB_column2`.
- Any non-missing values from the source columns are added to the corresponding new columns and are removed from the source column, since `pluck` is `true`.

## Syntax and Parameters

```
unnest col:column_ref keys:'key1','key2' [pluck:true|false] [markLineage:true|false]
```

| Token | Required? | Data Type | Description |
|-------|-----------|-----------|-------------|
| unnest | Y | transform | Name of the transform |
| | | | |

| col | Y | string | Source column name |
|---|---|---|---|
| keys | Y | string | Comma-separated list of quoted key names. See below for examples. |
| pluck | N | boolean | If `true`, any values unnested from the source are also removed from the source. Default is `false`. |
| markLineage | N | boolean | If `true`, the names of new columns are prepended with the name of the source column. |

For more information on syntax standards, see *Language Documentation Syntax Notes*.

**col**

Identifies the column to which to apply the transform. You can specify only one column.

**Usage Notes:**

| Required? | Data Type |
|---|---|
| Yes | String (column name) |

**keys**

Comma-separated list of keys to use to extract data from the specified source column.

- Key values must be quoted. (e.g `'key1'`,`'key2'`). Any quoted value is considered the path to a single key.
- Key values are case-sensitive.
- Each key must be listed. A range of keys cannot be specified.

> **NOTE:** Keys that contain non-alphanumeric values, such as spaces, must be enclosed in square brackets and quotes. Values with underscores do not require this bracketing.

The comma-separated list of keys determines the columns to generate from the source data. If you specify three values for keys, the three new columns contain the corresponding values from the source column.

This parameter has different syntax to use for single-level and multi-level nested data. There are also variations in syntax between Object and Array data type.

**Usage Notes:**

| Required? | Data Type |
|---|---|
| Yes | Comma-separated String values. Syntax examples are provided below. |

**Keys for Object data - single-level**

> **NOTE:** Key names are case-sensitive.

For a single, top-level key in an Object field, you can specify the key as a simple quoted string:

```
unnest col:myCol keys: 'myObjKey'
```

The above looks for the key `myObjKey` among the top-level keys in the Object and returns the corresponding value for the new column. You can also bracket this key in square brackets:

```
unnest col:myCol keys: '[myObjKey]'
```

To specify multiple first-level keys, use the following:

```
unnest col:myCol keys:'myObjKey','my2ndObjKey'
```

The above generates two new columns ( `myObjKey` and `my2ndObjKey`) containing the corresponding values for the keys.

**Keys for Object data - multi-level**

You can also reference keys that are below the first level in the Object.

Example data:

```
{ "Key1" :
  { "Key1A" :
    { "Key1A1" : "Value1" }
  }
}
{ "Key2" :
  { "Key2A" :
    { "Key2A1" : "Value2" }
  }
}
{ "Key3" :
  { "Key3A" :
    { "Key3A1" : "Value3" }
  }
}
```

To acquire the data for the `Key1A` key, use the following:

```
unnest col: myCol keys: 'Key1[Key1A]'
```

In the new column, the displayed value is the following:

```
{ "Key1A1" : "Value1" }
```

To unnest a third-layer value, use a transform similar to the following:

```
unnest col: myCol keys: 'Key2[Key2A][Key2A1]'
```

In the new column, this transform generates a value of `Value2`.

**Keys for Array data - single level**

You can reference array elements using zero-based indexes or key names.

> **NOTE:** All references to Array keys must be bracketed. Array keys can be referenced by index number only.

Example array data:

```
["red","orange","yellow","green","blue","indigo","violet"]
```

```
unnest col: myCol keys:'[1]'
```

The above transform retrieves the value `orange` from the array.

```
unnest col: myCol keys:'[1]','[3]'
```

Returned values: `orange` and `green`.

**Keys for Array data - multi-level**

The following example nested Array data matches the structure of the Object data in the previous example:

```
[ [ "Item1", ["Item1A", ["Item1A1","Value1"] ] ], [ "Item2", ["Item2A",  ["Item2A1","Value2"] ] ], [ "Item3",
["Item3A",["Item3A1","Value3"] ] ] ]
```

To unnest the value for `Items2A`:

```
unnest col:myCol keys:'[1][0]'
```

The value inserted into the new column is the following:

```
["Item2A1","Value2"]
```

To unnest from the third level:

```
unnest col:myCol keys:'[2][0][0]'
```

The inserted value is `Item3A`.

**pluck**

Indicates whether any values added from source to output columns should be removed from the source.

- Set to `true` to remove values from source after they have been added to output columns.
- (Default) Set to `false` to leave source columns untouched.

**Usage Notes:**

| Required? | Data Type |
|-----------|-----------|
| No | Boolean |

**markLineage**

When set to `true`, the names of new columns are prepended with the name of the source column. Example:

| Source Column | Output Column |
|---------------|---------------|
| mySourceColumn | mySourceColumn_column1 |

Nested key references are appended to the column name:

| Source Column | Key Value | Output Column |
|---------------|-----------|---------------|
| mySourceColumn | keys: '[Key1][Key2]' | mySourceColumn_Key1_Key2 |

> **NOTE:** If your `unnest` transform does not change the number of rows, you can still access source row number information in the data grid, assuming it was still available when the transform was executed.

**Usage Notes:**

| Required? | Data Type |
|-----------|-----------|
| No        | Boolean   |

## Examples

> **Tip:** For additional examples, see *Common Tasks*.

### Example - Unnest an Object

You have the following dataset. The `Sizes` column contains Object data on available sizes.

**Source:**

| ProdId | ProdName | Sizes |
|--------|----------|-------|
| 1001   | Hat      | {'Small':'N','Medium':'Y','Large':'Y','Extra-Large':'Y'} |
| 1002   | Shirt    | {'Small':'N','Medium':'Y','Large':'Y','Extra-Large':'N'} |
| 1003   | Pants    | {'Small':'Y','Medium':'Y','Large':'Y','Extra-Large':'N'} |

**Transformation:**

> **NOTE:** Depending on the format of your source data, you might need to perform some replacements in the `Sizes` column in order to make it inferred as proper Object type values. The final format should look like the above.

If it is not inferred already, set the type of the `Sizes` column to Object:

| Transformation Name | `Change column data type` |
|---------------------|--------------------------|
| Parameter: Columns  | `Sizes`                  |
| Parameter: New type | `Object`                 |

Unnest the data into separate columns. The following prepends `Sizes_` to the newly generated column name.

| Transformation Name | `Unnest Objects into columns` |
|---------------------|-------------------------------|
| Parameter: Column   | `Sizes`                       |
| Parameter: Paths to elements | `'Small','Medium','Large','Extra-Large'` |
| Parameter: Include original column name | `test` |

You might find it useful to add `pluck:true` to the above transform. When added, values that are un-nested are removed from the source, leaving only the values that weren't processed:

| Transformation Name | Unnest Objects into columns |
|---|---|
| Parameter: Column | Sizes |
| Parameter: Paths to elements | 'Small','Medium','Large','Extra-Large' |
| Parameter: Remove elements from original | true |
| Parameter: Include original column name | true |

If all values have been processed, the `Sizes` column now contains a set of maps missing data. You can use the following to determine if the length of the remaining data is longer than two characters. This transform is a good one to just preview:

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | (len(Sizes) > 2) |
| Parameter: New column name | 'len_Sizes' |

You can delete the source column:

| Transformation Name | Delete columns |
|---|---|
| Parameter: Columns | Sizes |
| Parameter: Action | Delete selected columns |

**Results:**

When you are finished, the dataset should look like the following:

| ProdId | ProdName | Sizes_Small | Sizes_Medium | Sizes_Large | Sizes_Extra-Large |
|---|---|---|---|---|---|
| 1001 | Hat | N | Y | Y | Y |
| 1002 | Shirt | N | Y | Y | N |
| 1003 | Pants | Y | Y | Y | N |

## Example - Unnest an array

The following example demonstrates differences between the `unnest` and the `flatten` transform, including how you use `unnest` to flatten array data based on specified keys.

- For more information, see *Flatten Transform*.

This example illustrates you to use the flatten and unnest transforms.

**Source:**

You have the following data on student test scores. Scores on individual scores are stored in the `Scores` array, and you need to be able to track each test on a uniquely identifiable row. This example has two goals:

1. One row for each student test
2. Unique identifier for each student-score combination

| LastName | FirstName | Scores |
|---|---|---|
| Adams | Allen | [81,87,83,79] |
| Burns | Bonnie | [98,94,92,85] |
| Cannon | Charles | [88,81,85,78] |

**Transformation:**

When the data is imported from CSV format, you must add a `header` transform and remove the quotes from the `Scores` column:

| Transformation Name | `Rename column with row(s)` |
|---|---|
| Parameter: Option | `Use row(s) as column names` |
| Parameter: Type | `Use a single row to name columns` |
| Parameter: Row number | `1` |

| Transformation Name | `Replace text or pattern` |
|---|---|
| Parameter: Column | `colScores` |
| Parameter: Find | `'\"'` |
| Parameter: Replace with | `''` |
| Parameter: Match all occurrences | `true` |

**Validate test date:** To begin, you might want to check to see if you have the proper number of test scores for each student. You can use the following transform to calculate the difference between the expected number of elements in the `Scores` array (4) and the actual number:

| Transformation Name | `New formula` |
|---|---|
| Parameter: Formula type | `Single row formula` |
| Parameter: Formula | `(4 - arraylen(Scores))` |
| Parameter: New column name | `'numMissingTests'` |

When the transform is previewed, you can see in the sample dataset that all tests are included. You might or might not want to include this column in the final dataset, as you might identify missing tests when the recipe is run at scale.

**Unique row identifier:** The `Scores` array must be broken out into individual rows for each test. However, there is no unique identifier for the row to track individual tests. In theory, you could use the combination of `LastName-FirstName-Scores` values to do so, but if a student recorded the same score twice, your dataset has duplicate

rows. In the following transform, you create a parallel array called `Tests`, which contains an index array for the number of values in the `Scores` column. Index values start at `0`:

| Transformation Name | New formula |
|---|---|
| **Parameter: Formula type** | Single row formula |
| **Parameter: Formula** | range(0,arraylen(Scores)) |
| **Parameter: New column name** | 'Tests' |

Also, we will want to create an identifier for the source row using the `sourcerownumber` function:

| Transformation Name | New formula |
|---|---|
| **Parameter: Formula type** | Single row formula |
| **Parameter: Formula** | sourcerownumber() |
| **Parameter: New column name** | 'orderIndex' |

**One row for each student test:** Your data should look like the following:

| LastName | FirstName | Scores | Tests | orderIndex |
|---|---|---|---|---|
| Adams | Allen | [81,87,83,79] | [0,1,2,3] | 2 |
| Burns | Bonnie | [98,94,92,85] | [0,1,2,3] | 3 |
| Cannon | Charles | [88,81,85,78] | [0,1,2,3] | 4 |

Now, you want to bring together the `Tests` and `Scores` arrays into a single nested array using the `arrayzip` function:

| Transformation Name | New formula |
|---|---|
| **Parameter: Formula type** | Single row formula |
| **Parameter: Formula** | arrayzip([Tests,Scores]) |

Your dataset has been changed:

| LastName | FirstName | Scores | Tests | orderIndex | column1 |
|---|---|---|---|---|---|
| Adams | Allen | [81,87,83,79] | [0,1,2,3] | 2 | [[0,81],[1,87],[2,83],[3,79]] |
| Adams | Bonnie | [98,94,92,85] | [0,1,2,3] | 3 | [[0,98],[1,94],[2,92],[3,85]] |
| Cannon | Charles | [88,81,85,78] | [0,1,2,3] | 4 | [[0,88],[1,81],[2,85],[3,78]] |

Use the following to unpack the nested array:

| Transformation Name | Expand arrays to rows |
|---|---|
| **Parameter: Column** | column1 |

Each test-score combination is now broken out into a separate row. The nested Test-Score combinations must be broken out into separate columns using the following:

| | |
|---|---|
| Transformation Name | Unnest Objects into columns |
| Parameter: Column | column1 |
| Parameter: Paths to elements | '[0]','[1]' |

After you delete `column1`, which is no longer needed you should rename the two generated columns:

| | |
|---|---|
| Transformation Name | Rename columns |
| Parameter: Option | Manual rename |
| Parameter: Column | column_0 |
| Parameter: New column name | 'TestNum' |

| | |
|---|---|
| Transformation Name | Rename columns |
| Parameter: Option | Manual rename |
| Parameter: Column | column_1 |
| Parameter: New column name | 'TestScore' |

**Unique row identifier:** You can do one more step to create unique test identifiers, which identify the specific test for each student. The following uses the original row identifier `OrderIndex` as an identifier for the student and the `TestNumber` value to create the `TestId` column value:

| | |
|---|---|
| Transformation Name | New formula |
| Parameter: Formula type | Single row formula |
| Parameter: Formula | (orderIndex * 10) + TestNum |
| Parameter: New column name | 'TestId' |

The above are integer values. To make your identifiers look prettier, you might add the following:

| | |
|---|---|
| Transformation Name | Merge columns |
| Parameter: Columns | 'TestId00','TestId' |

**Extending:** You might want to generate some summary statistical information on this dataset. For example, you might be interested in calculating each student's average test score. This step requires figuring out how to properly group the test values. In this case, you cannot group by the `LastName` value, and when executed at scale, there might be collisions between first names when this recipe is run at scale. So, you might need to create a kind of primary key using the following:

| | |
|---|---|
| Transformation Name | Merge columns |
| Parameter: Columns | 'LastName','FirstName' |
| Parameter: Separator | '-' |
| Parameter: New column name | 'studentId' |

You can now use this as a grouping parameter for your calculation:

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | average(TestScore) |
| Parameter: Group rows by | studentId |
| Parameter: New column name | 'avg_TestScore' |

**Results:**

After you delete unnecessary columns and move your columns around, the dataset should look like the following:

| TestId | LastName | FirstName | TestNum | TestScore | studentId | avg_TestScore |
|---|---|---|---|---|---|---|
| TestId0021 | Adams | Allen | 0 | 81 | Adams-Allen | 82.5 |
| TestId0022 | Adams | Allen | 1 | 87 | Adams-Allen | 82.5 |
| TestId0023 | Adams | Allen | 2 | 83 | Adams-Allen | 82.5 |
| TestId0024 | Adams | Allen | 3 | 79 | Adams-Allen | 82.5 |
| TestId0031 | Adams | Bonnie | 0 | 98 | Adams-Bonnie | 92.25 |
| TestId0032 | Adams | Bonnie | 1 | 94 | Adams-Bonnie | 92.25 |
| TestId0033 | Adams | Bonnie | 2 | 92 | Adams-Bonnie | 92.25 |
| TestId0034 | Adams | Bonnie | 3 | 85 | Adams-Bonnie | 92.25 |
| TestId0041 | Cannon | Chris | 0 | 88 | Cannon-Chris | 83 |
| TestId0042 | Cannon | Chris | 1 | 81 | Cannon-Chris | 83 |
| TestId0043 | Cannon | Chris | 2 | 85 | Cannon-Chris | 83 |
| TestId0044 | Cannon | Chris | 3 | 78 | Cannon-Chris | 83 |

**Example - extracting key values from car data and then unnesting into separate columns**

This example shows how you can unpack data nested in an Object into separate columns.

**Source:**

You have the following information on used cars. The `VIN` column contains vehicle identifiers, and the `Properti es` column contains key-value pairs describing characteristics of each vehicle. You want to unpack this data into separate columns.

| VIN | Properties |
|---|---|
| XX3 JT4522 | year=2004,make=Subaru,model=Impreza,color=green,mileage=125422,cost=3199 |
| HT4 UJ9122 | year=2006,make=VW,model=Passat,color=silver,mileage=102941,cost=4599 |
| KC2 WZ9231 | year=2009,make=GMC,model=Yukon,color=black,mileage=68213,cost=12899 |
| LL8 UH4921 | year=2011,make=BMW,model=328i,color=brown,mileage=57212,cost=16999 |

**Transformation:**

Add the following transformation, which identifies all of the key values in the column as beginning with alphabetical characters.

- The `valueafter` string identifies where the corresponding value begins after the key.
- The `delimiter` string indicates the end of each key-value pair.

| | |
|---|---|
| **Transformation Name** | `Convert keys/values into Objects` |
| **Parameter: Column** | `Properties` |
| **Parameter: Key** | `` `{alpha}+` `` |
| **Parameter: Separator between key and value** | `` `=` `` |
| **Parameter: Delimiter between pair** | `','` |

Now that the Object of values has been created, you can use the `unnest` transform to unpack this mapped data. In the following, each key is specified, which results in separate columns headed by the named key:

> **NOTE**: Each key must be entered on a separate line in the Path to elements area.

| | |
|---|---|
| **Transformation Name** | `Unnest Objects into columns` |
| **Parameter: Column** | `extractkv_Properties` |
| **Parameter: Paths to elements** | `year` |
| **Parameter: Paths to elements** | `make` |
| **Parameter: Paths to elements** | `model` |
| **Parameter: Paths to elements** | `color` |
| **Parameter: Paths to elements** | `mileage` |
| **Parameter: Paths to elements** | `cost` |

**Results:**

When you delete the unnecessary Properties columns, the dataset now looks like the following:

| VIN | year | make | model | color | mileage | cost |
|---|---|---|---|---|---|---|
| XX3 JT4522 | 2004 | Subaru | Impreza | green | 125422 | 3199 |
| HT4 UJ9122 | 2006 | VW | Passat | silver | 102941 | 4599 |
| KC2 WZ9231 | 2009 | GMC | Yukon | black | 68213 | 12899 |
| LL8 UH4921 | 2011 | BMW | 328i | brown | 57212 | 16999 |

# Unpivot Transform

**Contents:**

---

> **NOTE:** Transforms are a part of the underlying language, which is not directly accessible to users. This content is maintained for reference purposes only. For more information on the user-accessible equivalent to transforms, see *Transformation Reference*.

Reshapes the data by merging one or more columns into key and value columns. Keys are the names of input columns, and value columns are the cell values from the source.

Rows of data are duplicated, once for each input column.

The `unpivot` column can be applied to multiple columns in the same transform. All columns are un-pivoted into the same `key` and `value` columns. When this transform is applied to two columns, the number of rows in the dataset is doubled.

This transform is the opposite of the `pivot` transform, which converts a set of column values into distinct columns. See *Pivot Transform* .

## Basic Usage

**Single- or multi-column example:**

You can specify single columns or comma-separated sets of columns.

```
unpivot col: FirstName, MiddleInitial
```

**Output:** Converts the values in the columns `FirstName` and `MiddleInitial` into separate `key` and `value` columns.

**Column range example:**

You can also specify ranges of columns using the tilde (~) operator:

```
unpivot col:Column1~Column20
```

**Output:** Converts all of the values in columns between `Column1` and `Column20` into `key` and `value` columns.

## Syntax and Parameters

```
unpivot col: column_ref [groupEvery: int_num]
```

| Token | Required? | Data Type | Description |
|-------|-----------|-----------|-------------|
| unpivot | Y | transform | Name of the transform |
| col | Y | string | Name of source column or columns |
| groupEvery | N | string | If specified, this parameter defines the number of individual key-value pairs to store in each generated column. Default is 1. |

For more information on syntax standards, see *Language Documentation Syntax Notes*.

**col**

Identifies the column or columns to which to apply the transform. You can specify one or more columns.

To specify multiple columns:

- Discrete column names are comma-separated.
- Values for column names are case-sensitive.

- Column ranges are supported:

```
myColumn1~myColumn5
```

**NOTE:** For the `col` value, you can use the asterisk ( * ) wildcard to apply the unpivot to the entire dataset, which generates a `key` and a `value` column, containing all column-row entries from the source columns. However, unpivoting a large number of columns can significantly increase the number of rows in your dataset.

**Usage Notes:**

| Required? | Data Type |
|-----------|-----------|
| Yes | String (column name) |

**groupEvery**

Specifies the number of output key-value pair columns to produce after unpivoting.

This optional parameter is used to create multiple sets of key-value pair columns in the output. The columns listed in the `col` parameter are placed into each pair of output key-value columns sequentially. After all key-value pair columns are filled in a record, the next column is placed into the first key-value pair column of the next record.

By default, this value is 1, meaning that each column specified in the transform is rendered into a new record in a single pair of key-value columns.

**Usage Notes:**

| Required? | Data Type |
|-----------|-----------|
| No | Integer (positive) |

# Examples

> **Tip:** For additional examples, see *Common Tasks*.

**Example - Basic Unpivot**

**Source:**

| productName | productColor | productSize |
|---|---|---|
| Whizbang | red | M |
| Whizbang | red, blue | L |
| Whizbang | green | M |
| Bangwhiz | red | S |
| Bangwhiz | blue | M |
| Bangwhiz | red | S |

**Tranformation:**

After you have created a header, if necessary, add the following transformation:

| Transformation Name | `Unpivot columns` |
|---|---|
| Parameter: Columns | `productColor` |

**Results:**

| productName | productSize | key | value |
|---|---|---|---|
| Whizbang | M | productColor | red |
| Whizbang | L | productColor | red, blue |
| Whizbang | M | productColor | green |
| Bangwhiz | S | productColor | red |
| Bangwhiz | M | productColor | blue |
| Bangwhiz | S | productColor | red |

**Extended:**

Note how each instance of a value results in a separate row; duplicate values are included. For a single-column `unpivot`, this transform results in the same number of rows as the source.

- Since the value is treated as a string, the value `red, blue` is treated as one value.

Now, edit the transformation you just added. Replace it with the following, which includes the `productSize` key as part of the transformation:

| Transformation Name | `Unpivot columns` |
|---|---|
| Parameter: Columns | `productColor,productSize` |

**Results:**

| productName | key | value |
|---|---|---|
| Whizbang | productColor | red |
| Whizbang | productSize | M |
| Whizbang | productColor | red, blue |
| Whizbang | productSize | L |
| Whizbang | productColor | green |
| Whizbang | productSize | M |
| Bangwhiz | productColor | red |
| Bangwhiz | productSize | S |
| Bangwhiz | productColor | blue |
| Bangwhiz | productSize | M |
| Bangwhiz | productColor | red |
| Bangwhiz | productSize | S |

Row keys alternate based on the order in which the source columns are specified in the transform. Since the transform specifies two columns, the number of key-value pairs is doubled, which results in a dataset that has twice as many rows as the source.

## Example - Basic Pivot with groupEvery

**Tranformation:**

From the previous example, modify the `unpivot` transform to be the following:

| Transformation Name | `Unpivot columns` |
|---|---|
| **Parameter: Columns** | `productColor,productSize` |
| **Parameter: Group size** | `2` |

**Results:**

| productName | key1 | value1 | key2 | value2 |
|---|---|---|---|---|
| Whizbang | productColor | red | productSize | M |
| Whizbang | productColor | red, blue | productSize | L |
| Whizbang | productColor | green | productSize | M |
| Bangwhiz | productColor | red | productSize | S |
| Bangwhiz | productColor | blue | productSize | M |
| Bangwhiz | productColor | red | productSize | S |

# Valuestocols Transform

**Contents:**

---

> **NOTE:** Transforms are a part of the underlying language, which is not directly accessible to users. This content is maintained for reference purposes only. For more information on the user-accessible equivalent to transforms, see *Transformation Reference*.

For each unique value in a column, a separate column is created. For each row that contains the value in the source column, an indicator value is inserted in the new column.

- This indicator value can be a literal value or the output of a function.
- If no indicator value is generated, a null value is written.

This transform is used to generate indicator columns, which can be used in statistical analysis.

- It evaluates entire cell values for uniqueness. It does not scan for individual elements in Object or Array data.
- If a row in the source column contains a missing value, an indicator value is added in a new `Empty` column.
- It is not appropriate for tabulating counts of strings or patterns in a column. See *Countpattern Transform* .

Optionally, you can specify a default value, which is applied to all non-indicator value cells in the new column.

> **NOTE:** When this transform is applied in the data grid, it only identifies the unique values in the current sample. If there are other unique values in the entire dataset, new columns are not created for them when the transform is executed across the entire dataset.

## Basic Usage

**Source:**

| Data |
| --- |
| Happy |
| Dog |
| Happy Happy Dog |

**Transformation:**

```
valuestocols col:Data value:'X'
```

**Results:**

| Data | Happy | Dog | Happy_Happy_Dog |
|---|---|---|---|
| Happy | X | | |
| Dog | | X | |
| Happy Happy Dog | | | X |

## Syntax and Parameters

```
valuestocols col:column_ref value:(expression) default:(expression) [limit:int_num]
```

| Token | Required? | Data Type | Description |
|---|---|---|---|
| values tocols | Y | transform | Name of the transform |
| col | Y | string | Name of source column |
| value | Y | string | String literal, column, or function call that defines the value to use as the indicator value in any newly generated column |
| default | N | string | String literal, column, or function call that defines the value to use to indicate a false match in any newly generated column |
| limit | N | integer (positive) | Maximum number of columns to generate. Default is $50$. |

For more information on syntax standards, see *Language Documentation Syntax Notes*.

**col**

Identifies the column to which to apply the transform. You can specify only one column.

**Usage Notes:**

| Required? | Data Type |
|---|---|
| Yes | String (column name) |

**value**

For the `valuestocols` transform, this parameter specifies the value to insert in each row of a generated column where the column name of the generated column appears in the same row of the source column. This value can be a string literal, a column reference, or a function.

**Usage Notes:**

| Required? | Data Type |
|---|---|
| Yes | String literal, column reference, or function call |

**default**

Optionally, this parameter can be used to specify the value to insert in each row of a generated column where the column name of the generated column does not appear in the same row of the source column. This value can be a string literal, a column reference, or a function.

If this parameter is not specified, a missing value is inserted.

**Usage Notes:**

| Required? | Data Type |
|---|---|
| No | String literal, column reference, or function call |

### limit

The `limit` parameter defines the maximum number of columns to create from the unique values detected in the source column. If not specified, the limit is `50`.

> **NOTE:** Be careful setting this parameter too high. In some cases, the application can run out of memory generating the results, and your results can fail.

**Usage Notes:**

| Required? | Data Type |
|---|---|
| No. Default value is `50`. | Integer (positive) |

## Examples

> **Tip:** For additional examples, see *Common Tasks*.

### Example - Basic valuestocols

**Source:**

This dataset contains milestones for three employees who joined the company at the same time. The milestones were recorded and organized by date as individual items, so it's not easy to verify that all five milestones have been checked off for each employee:

- Orientation
- Contact Info
- Acquire Computer
- HR Policies Training
- Product Training

| Date | Name | Checklist |
|---|---|---|
| 4/4/16 | Bowie Kuhn | Orientation |
| 4/4/16 | Happy Chandler | Contact Info |
| 4/4/16 | Bowie Kuhn | Contact Info |
| 4/4/16 | Bowie Kuhn | Acquire Computer |
| 4/4/16 | Bud Selig | Product Training |
| 4/4/16 | Bud Selig | Orientation |
| 4/5/16 | Happy Chandler | HR Policies Training |
| 4/5/16 | Happy Chandler | Orientation |

| 4/5/16 | Happy Chandler | Acquire Computer |
| 4/5/16 | Bowie Kuhn | HR Policies Training |
| 4/5/16 | Bud Selig | HR Policies Training |
| 4/5/16 | Bud Selig | Contact Info |
| 4/6/16 | Happy Chandler | Product Training |

### Transformation:

The following transform creates columns for each of the values in the `Checklist` column and adds a `yes` value where there is a match for the row:

| Transformation Name | `Convert values to columns` |
| --- | --- |
| **Parameter: Column** | `onboardingChecklist` |
| **Parameter: Fill when present** | `'yes'` |

### Results:

In the generated columns, you can quickly assess whether all three employees have completed an individual checklist item:

- Bud Selig has not acquired his computer.
- Bowie Kuhn has not had product training.

| Date | Name | Checklist | Orientation | Contact_Info | Acquire_Computer | Product_Training | HR_Policies_Training |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 4/4/16 | Bowie Kuhn | Orientation | yes | | | | |
| 4/4/16 | Happy Chandler | Contact Info | | yes | | | |
| 4/4/16 | Bowie Kuhn | Contact Info | | yes | | | |
| 4/4/16 | Bowie Kuhn | Acquire Computer | | | yes | | |
| 4/4/16 | Bud Selig | Product Training | | | | yes | |
| 4/4/16 | Bud Selig | Orientation | yes | | | | |
| 4/5/16 | Happy Chandler | HR Policies Training | | | | | yes |
| 4/5/16 | Happy Chandler | Orientation | yes | | | | |
| 4/5/16 | Happy Chandler | Acquire Computer | | | yes | | |
| 4/5/16 | Bowie Kuhn | HR Policies Training | | | | | yes |
| 4/5/16 | Bud Selig | HR Policies Training | | | | | yes |
| 4/5/16 | Bud Selig | Contact Info | | yes | | | |
| 4/6/16 | Happy Chandler | Product Training | | | | yes | |

**Example - Magazine subscriptions**

This example shows how you can break out a column of nested values into separate rows and columns of data.

**Source:**

The following data covers magazine subscriptions for individual customers. Their subscriptions are stored in an array of values. You are interested in who is subscribing to each magazine.

| CustId | Subscriptions |
|--------|---------------|
| Anne Aimes | ["Little House and Garden","Sporty Pants","Life on the Range"] |
| Barry Barnes | ["Sporty Pants","Investing Smart"] |
| Cindy Compton | ["Cakes n Pies","Powerlifting Plus","Running Days"] |
| Darryl Diaz | ["Investing Smart","Cakes n Pies"] |

**Transformation:**

When this data is loaded into the Transformer, you might need to apply a `header` to it. If it is in CSV format, you might need to apply some `replace` transformations to clean up the `Subscriptions` column so it looks like the above.

When the `Subscriptions` column contains cleanly formatted arrays, the column is re-typed as Array type. You can then apply the following transformation:

| Transformation Name | `Expand Array into rows` |
|---------------------|--------------------------|
| Parameter: Column | `Subscriptions` |

Each `CustId`/`Subscription` combination is now written to a separate row. You can use this new data structure to break out instances of magazine subscriptions. Using the following transformation, you can add the corresponding `CustId` value to the column:

| Transformation Name | `Convert values to columns` |
|---------------------|------------------------------|
| Parameter: Column | `Subscriptions` |
| Parameter: Fill when present | `CustId` |

Delete the two source columns:

| Transformation Name | `Delete columns` |
|---------------------|-------------------|
| Parameter: Columns | `CustId,Subscriptions` |
| Parameter: Action | `Delete selected columns` |

**Results:**

| Little_House_and_Garden | Sporty_Pants | Life_on_the_Range | Investing_Smart | Cakes_n_Pies | Powerlifting_Plus | R |
|-------------------------|--------------|--------------------|------------------|--------------|---------------------|---|
| Anne Aimes | | | | | | |
| | Anne Aimes | | | | | |
| | | Anne Aimes | | | | |

| | Barry Barnes | | | | | |
| | | | Barry Barnes | | | |
| | | | | Cindy Compton | | |
| | | | | | Cindy Compton | |
| | | | | | | Cin |
| | | | Darryl Diaz | | | |
| | | | | Darry Diaz | | |

# Window Transform

**Contents:**

---

> **NOTE:** Transforms are a part of the underlying language, which is not directly accessible to users. This content is maintained for reference purposes only. For more information on the user-accessible equivalent to transforms, see *Transformation Reference.*

The `window` transform enables you to perform summations and calculations based on a rolling window of data relative to the current row.

For example, you can compute the rolling average for a specified column for the current row value and the three preceding rows. This transform is particularly useful for processing time or otherwise sequential data.

You can apply one or more functions to your `window` transform step.

- For more information on window functions, see *Window Functions.*
- You can also use the aggregation functions with this transform. See *Aggregate Functions.*

> **NOTE:** Be careful applying this transform across a large number of rows. In some cases, the application can run out of memory generating the results, and your results can fail.

## Basic Usage

```
window value: ROLLINGAVERAGE(myValues,3) order: MyDate group: customerId
```

**Output:** Generates a new column called, `window`, which contains the result of the `ROLLINGAVERAGE` function applied from the current row in the `myValues` column across the `3` rows forward, ordered by `MyDate` and grouped by `customerId`.

## Syntax and Parameters

```
window value: WINDOW_FUNCTION(arg1,arg2) order: order_col [group: group_col]
```

| Token | Required? | Data Type | Description |
|-------|-----------|-----------|-------------|
| window | Y | transform | Name of the transform |
| value | Y | string | Expression that evaluates to the window function call and its parameters |
| order | Y | string | Column or column names by which to sort the dataset before the `value` expression is applied |
| group | N | string | Column name or names containing the values by which to group for calculation |

For more information on syntax standards, see *Language Documentation Syntax Notes.*

### value

For the `window` transform, the `value` parameter contains the function call or calls, which define the set of rows to which the function is applied.

You can specify multiple window functions for the value. Each function reference must be separated by a comma. The transform generates a new column for each window function.

This transform uses a special set of functions. For more information on the available functions, see *Window Functions*.

**Usage Notes:**

| Required? | Data Type |
|---|---|
| Yes | String (expression) |

### order

For the `window` transform, this parameter specifies the column on which to sort the dataset before applying the specified function. For combination sort keys, you can add multiple comma-separated columns.

> **NOTE:** The order parameter must unambiguously specify an ordering for the data, or the generated results may vary between job executions.

> **NOTE:** If you are applying a window function, it requires a primary key to identify rows in the output. Otherwise, results can be ambiguous. For more information on defining a primary key, see *Window Functions*.

> **NOTE:** If it is present, the dataset is first grouped by the `group` value before it is ordered by the values in the `order` column.

> **NOTE:** The `order` column does not need to be sorted before the `window` transform is executed on it.

> **Tip:** To sort in reverse order, prepend the column name with a dash (`-MyDate`).

**Usage Notes:**

| Required? | Data Type |
|---|---|
| Yes | String (column name) |

### group

For the `window` transform, this parameter specifies the column whose values are used to group the dataset prior to applying the specified function. For combination grouping, you can specify multiple comma-separated column names.

> **NOTE:** Transforms that use the `group` parameter can result in non-deterministic re-ordering in the data grid. However, you should apply the `group` parameter, particularly on larger datasets, or your job may run out of memory and fail. To enforce row ordering, you can use the `sort` transform. For more information, see *Sort Transform*.

**Usage Notes:**

| Required? | Data Type |
|-----------|-----------|
| No | String (column name) |

## Examples

> **Tip:** For additional examples, see *Common Tasks*.

See the individual functions for examples. See *Window Functions*.

# Transformation Examples

This section contains examples that demonstrate how Wrangle transformations and functions can quickly transform your data.

> **NOTE:** These examples appear in other pages, including language reference documentation. See *Wrangle Language.*

## *Transformation Examples*

| Item | Description |
|------|-------------|
| *EXAMPLE - ARRAYINDEXOF and ARRAYRIGHTINDEXOF Functions* | This example illustrates how to convert the index value of an array for a specified value searching from left to right and right to left by using ARRAYINDEXOF and ARRAYRIGTHINDEXOF functions. |
| *EXAMPLE - ARRAYLEN and ARRAYELEMENTAT Functions* | This example illustrates how to return n-based number of elements in an array. |
| *EXAMPLE - ARRAYSLICE and ARRAYMERGEELEMENTS Functions* | This example illustrates how to generate an Array that is a slice of an another Array, based on index numbers. The elements of this Array can then be merged into a String value. |
| *EXAMPLE - ARRAYSTOMAP Function* | This example illustrates how to use the ARRAYSTOMAP and KEYS functions to convert values in Array or Object data type of key-value pairs. |
| *EXAMPLE - Base64 Encoding Functions* | This example demonstrates how to convert an input string to a base64-encoded value and back to ASCII text strings. |
| *EXAMPLE - Case Functions* | This example demonstrates functions that can be used to change the case of String values. |
| *EXAMPLE - Comparison Functions1* | This example illustrates the comparison functions in Designer Cloud Powered by Trifacta® Enterprise Edition. |
| *EXAMPLE - Comparison Functions2* | This example demonstrates functions for comparing the relative values of two functions. |
| *EXAMPLE - Comparison Functions Equal* | This example demonstrates comparison functions. |
| *EXAMPLE - Conditional Calculations Functions* | This example illustrates how to use the conditional calculation functions. |
| *EXAMPLE - COUNT Functions* | This example demonstrates how to count the number of values and non-null values within a group. |
| *EXAMPLE - COUNTIF Functions* | This example demonstrates how to count the number of values within a group, based on a specified conditional test. |
| *EXAMPLE - Countpattern Transform* | This example demonstrates how to count the number of occurrences of text patterns in a column. |
| *EXAMPLE - DATE and TIME Functions* | This example provides an overview on various date and time functions. |
| *EXAMPLE - Date Difference Functions* | This example demonstrates how to calculate the number of days between two input dates. |
| *EXAMPLE - DATEDIF Function* | This example illustrates how to calculate the number of days that have elapsed between the order date and today. |
| *EXAMPLE - Date Functions* | This example illustrates to how to use date-related functions to derive specific values for a Datetime column type. |
| *EXAMPLE - Date Functions - Min Max and Mode* | This example shows how you can apply statistical functions on Datetime columns. |
| *EXAMPLE - DATEIF Functions* | This example illustrates how you can apply conditionals to calculate minimum, maximum, and most common date values. |

| | |
|---|---|
| *EXAMPLE - Day of Functions* | This example illustrates how you can apply functions to derive day-of-week values out of a column of Datetime type. |
| *EXAMPLE - DEGREES and RADIANS Functions* | This example illustrates to convert values from one unit of measure to the other. |
| *EXAMPLE - Delete and Keep Transforms* | This examples illustrates how you can keep and delete rows from your dataset. |
| *EXAMPLE - Domain Functions* | This examples illustrates how you can extract component parts of a URL using specialized functions for the URL data type. |
| *EXAMPLE - Double Metaphone Functions* | This example illustrates how to use double metaphone functions to generate phonetic spellings in Designer Cloud Powered by Trifacta® Enterprise Edition. |
| *EXAMPLE - Exponentional Functions* | This example demonstrates the exponential functions. |
| *EXAMPLE - Extractkv and Unnest Transforms* | This example shows how you can unpack data nested in an Object into separate columns. |
| *EXAMPLE - Extractlist Transform* | This example illustrates how to extract values from a column. |
| *EXAMPLE - Extract Values* | In this example, you extract one or more values from a source column and assemble them in an Array column. |
| *EXAMPLE - Flatten an Array* | This section describes how to flatten the values in an Array into separate rows in your dataset. |
| *EXAMPLE - Flatten and Unnest Transforms* | This example illustrates you to use the flatten and unnest transforms. |
| *EXAMPLE - Flatten and Valuestocols Transforms* | This example shows how you can break out a column of nested values into separate rows and columns of data. |
| *EXAMPLE - IF Data Type Validation Functions* | This example illustrates how to use the IF* functions for data type validation. |
| *EXAMPLE - IPTOINT Function* | This examples illustrates how you can convert IP addresses to numeric values for purposes of comparison and sorting. |
| *EXAMPLE - KTHLARGESTDATE Functions* | This example illustrates how you can apply conditionals to calculate minimum, maximum, and most common date values. |
| *EXAMPLE - KTHLARGEST Function* | This example explores how you can use aggregation functions to calculate rank of values in a column. |
| *EXAMPLE - KTHLARGESTIF Function* | This example illustrates how to use the conditional ranking functions. |
| *EXAMPLE - LIST and UNIQUE Function* | This example demonstrates you to extract values from one column of an array into a new column. |
| *EXAMPLE - LISTIF Functions* | This example illustrates you to identify and list all values within a group that meet a specified condition. |
| *EXAMPLE - LIST Math Functions* | This example describes how to generate random array (list) data and then to apply statistical functions specifically created for arrays. |
| *EXAMPLE - Logical Functions* | This example demonstrate the AND, OR, and NOT logical functions. |
| *EXAMPLE - Nest and Unnest JSON Records* | This example illustrates how you can use unnesting and nesting transformations to reshape your JSON data. |
| *EXAMPLE - Nest Columns into Arrays* | This section provides simple examples of nesting columns into Arrays by extracting values from a column or nesting one or more columns into an Array column. |
| *EXAMPLE - Nest Columns into Objects* | This section provides a simple example of nesting columns into a new column of Object data type. |
| *EXAMPLE - Nested Functions* | This example illustrates how to use the nested functions. |
| *EXAMPLE - Nest JSON Records* | This section illustrates a simple example of how to nest tabular data into JSON records. |
| *EXAMPLE - NEXT Function* | This example covers how to use the NEXT function to create windows of data from the current row and subsequent (next) rows in the dataset. You can then apply rolling computations across these windows of data. |
| *EXAMPLE - NOW and TODAY Functions* | This example illustrates you to generate the date and time values for the current date and timestamp in the specified time zone. |

| | |
|---|---|
| *EXAMPLE - Numeric Functions* | This example demonstrates how to use numeric functions to perform computations in your recipe steps. |
| *EXAMPLE - Percentile Functions* | This example illustrates you to apply percentile functions. |
| *EXAMPLE - POW and SQRT Functions* | In this example, you learn how to compute exponentials and square roots on your numeric data. |
| *EXAMPLE - PREV Function* | This example describes how you can use the PREV function to analyze data that is available in a window in rows before the current one. |
| *EXAMPLE - Quote Parameter* | This example demonstrates how to use `quote` parameter for more sophisticated splitting of columns of data using the `split` transform. |
| *EXAMPLE - RANDBETWEEN and PI Functions* | This example illustrates how you can apply functions to generate random numeric data in your dataset. |
| *EXAMPLE - RANK Functions* | This example demonstrates you to generate a ranked order of values. |
| *EXAMPLE - Replacement Transforms* | This example illustrates the different uses of the replacement transformations to replace or extract cell data. |
| *EXAMPLE - Rolling Date Functions* | This example describes how to use rolling functions for Datetime values. |
| *EXAMPLE - Rolling Functions* | This example describes how to use rolling computational functions. |
| *EXAMPLE - Rolling Functions 2* | This example describes how to use rolling statistical functions. |
| *EXAMPLE - ROLLINGKTHLARGEST Functions* | This example describes how to use rolling kthlargest functions for calculating ranking of values within a defined window of rows. |
| *EXAMPLE - Rounding Functions* | This example demonstrates how the rounding functions work together. |
| *EXAMPLE - Change data type transformation* | This example illustrates how to clean up data by changing its data type to String, manipulating it using String functions, and then retyping the data to its proper data type. |
| *EXAMPLE - Rename columns with rows* | This example illustrates how you can rename columns based on the contents of specified rows. |
| *EXAMPLE - Splitting with Different Delimiter Types* | This example shows how you can split data from a single column into multiple columns using delimiters. |
| *EXAMPLE - STARTSWITH and ENDSWITH Functions* | This example demonstrates functions that can be used to evaluate the beginning and end of values of any type using patterns. |
| *EXAMPLE - Statistical Functions* | This example illustrates how you can apply statistical functions to your dataset. Calculations include average (mean), max, min, standard deviation, and variance. |
| *EXAMPLE - Statistical Functions Sample Method* | This example shows some of the statistical functions that use the sample method of computation. |
| *EXAMPLE - String Cleanup Functions* | This example demonstrates functions that can be used to clean up strings. |
| *EXAMPLE - String Comparison Functions* | This example demonstrates functions that can be used to compare two sets of strings. |
| *EXAMPLE - SUMIF and COUNTDISTINCTIF Functions* | This example illustrates how you can use conditional calculation functions. |
| *EXAMPLE - SUMIF Function* | This example can be used to sum the values in a column based on a condition and organized by group. |
| *EXAMPLE - Time Zone Conversion Functions* | This example shows how you can use functions to convert Datetime values to different time zones. |
| *EXAMPLE - Trigonometry Arc Functions* | This example illustrates how to apply the inverse trigonometric (Arc) functions to your transformations. |
| *EXAMPLE - Trigonometry Functions* | This example illustrates how to apply basic trigonometric functions to your transformations. |
| *EXAMPLE - Trigonometry Hyperbolic Arc Functions* | This example illustrates how to apply inverse (arc) hyperbolic functions to your transformations. |
| *EXAMPLE - Trigonometry Hyperbolic Functions* | This example illustrates how to apply hyperbolic trigonometric functions to your transformations. All of the functions take inputs in radians. |

| | |
|---|---|
| *EXAMPLE - Two-Column Statistical Functions* | This example illustrates statistical functions that can be applied across two columns of values. |
| *EXAMPLE - Type Functions* | This example illustrates how various type checking functions can be applied to your data. |
| *EXAMPLE - Type Parsing Functions* | This example shows how to use parsing functions for evaluating input values against the function-specific data type. |
| *EXAMPLE - UNICODE Function* | In this example, you can see how the `CHAR` function can be used to convert numeric index values to Unicode characters, and the `UNICODE` function can be used to convert characters back to numeric values. |
| *EXAMPLE - Unixtime Functions* | This example illustrates how you can use functions to manipulate Unix time values in a column of Datetime type. |
| *EXAMPLE - Unnest an Array* | This section describes how to unnest the values in an Array into separate columns in your dataset. |
| *EXAMPLE - Unnest JSON Records* | You can unnest a set of JSON records into new columns of tabular data for easier manipulation within the application. |
| *EXAMPLE - Extract Keys From Objects* | You can extract the keys from an Object column into an Array of String values. |
| *EXAMPLE - Extract Object Values* | This simple example demonstrates how to extract nested values from Object elements into a separate column. |
| *EXAMPLE - Filtering Strings into Objects* | You can create nested objects by filtering strings. In this example, column headers and column values are nested into a single entity in a new column of Object data type. |

# EXAMPLE - ARRAYINDEXOF and ARRAYRIGHTINDEXOF Functions

This example illustrates how to convert the index value of an array for a specified value searching from left to right and right to left by using ARRAYINDEXOF and ARRAYRIGTHINDEXOF functions.

**Functions**:

| Item | Description |
|---|---|
| ARRAYINDEXOF Function | Computes the index at which a specified element is first found within an array. Indexing is left to right. |
| ARRAYRIGHTINDEXOF Function | Computes the index at which a specified element is first found within an array, when searching right to left. Returned value is based on left-to-right indexing. |

**Source:**

The following set of arrays contain results, in order, of a series of races. From this list, the goal is to generate the score for each racer according to the following scoring matrix.

| Place | Points |
|---|---|
| 1st | 30 |
| 2nd | 20 |
| 3rd | 10 |
| Last | -10 |
| Did Not Finish (DNF) | -20 |

Results:

| RaceId | RaceResults |
|---|---|
| 1 | ["racer3","racer5","racer2","racer1","racer6"] |
| 2 | ["racer6","racer4","racer2","racer1","racer3","racer5"] |
| 3 | ["racer4","racer3","racer5","racer2","racer6","racer1"] |
| 4 | ["racer1","racer2","racer3","racer5"] |
| 5 | ["racer5","racer2","racer4","racer6","racer3"] |

**Transformation:**

Note that the number of racers varies with each race, so determining the position of the last racer depends on the number in the event. The number of racers can be captured using the following:

| | |
|---|---|
| **Transformation Name** | New formula |
| **Parameter: Formula type** | Single row formula |
| **Parameter: Formula** | ARRAYLEN(RaceResults) |
| **Parameter: New column name** | 'countRacers' |

Create columns containing the index values for each racer. Below is the example for `racer1`:

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | ARRAYINDEXOF(RaceResults, 'racer1') |
| Parameter: New column name | 'arrL-IndexRacer1' |

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | ARRAYRIGHTINDEXOF(RaceResults, 'racer1') |
| Parameter: New column name | 'arrR-IndexRacer1' |

You can then compare the values in the two columns to determine if they are the same.

> **NOTE:** If ARRAYINDEXOF and ARRAYRIGHTINDEXOF do not return the same value for the same inputs, then the value is not unique in the array.

Since the points awarded for 1st, 2nd, and 3rd place follow a consistent pattern, you can use the following single statement to compute points for podium finishes for `racer1`: computing based on the value stored for the left index value:

| Transformation Name | Conditional column |
|---|---|
| Parameter: Condition type | if...then...else |
| Parameter: If | {arrayL-IndexRacer1} < 3 |
| Parameter: Then | (3 - {arrayL-IndexRacer1}) * 10 |
| Parameter: Else | 0 |
| Parameter: New column name | 'ptsRacer1' |

The following transform then edits the `ptsRacer1` to evaluate for the Did Not Finish (DNF) and last place conditions:

| Transformation Name | Edit column with formula |
|---|---|
| Parameter: Columns | ptsRacer1 |
| Parameter: Formula | IF(ISNULL({arrayL-IndexRacer1}), -20, ptsRacer1)) |

You can use the following to determine if the specified racer was last in the event:

| Transformation Name | Edit column with formula |
|---|---|
| Parameter: Columns | ptsRacer1 |
| Parameter: Formula | IF(arrR-IndexRacer1 == countRacers, -10, ptsRacer1) |

**Results:**

| RaceId | RaceResults | countRacers | arrR-IndexRacer1 | arrL-IndexRacer1 | ptsRacer1 |
|---|---|---|---|---|---|
| 1 | ["racer3","racer5","racer2","racer1","racer6"] | 5 | 3 | 3 | 0 |
| 2 | ["racer6","racer4","racer2","racer1","racer3","racer5"] | 6 | 3 | 3 | 0 |
| 3 | ["racer4","racer3","racer5","racer2","racer6","racer1"] | 6 | 5 | 5 | -10 |
| 4 | ["racer1","racer2","racer3","racer5"] | 4 | 0 | 0 | 20 |
| 5 | ["racer5","racer2","racer4","racer6","racer3"] | 5 | null | null | -20 |

# EXAMPLE - ARRAYLEN and ARRAYELEMENTAT Functions

This example illustrates how to return n-based number of elements in an array.

**Functions**:

| Item | Description |
|------|-------------|
| ARRAYLEN Function | Computes the number of elements in the arrays in the specified column, array literal, or function that returns an array. |
| ARRAYELEMENTAT Function | Computes the 0-based index value for an array element in the specified column, array literal, or function that returns an array. |
| ARRAYSORT Function | Sorts array values in the specified column, array literal, or function that returns an array in ascending or descending order. |

**Source:**

Here are some student test scores. Individual scores are stored in the `Scores` column. You want to:

1. Flag the students who have not taken four tests.
2. Compute the range in scores for each student.

| LastName | FirstName | Scores |
|----------|-----------|--------|
| Allen | Amanda | [79, 83,87,81] |
| Bell | Bobby | [85, 92, 94, 98] |
| Charles | Cameron | [88,81,85] |
| Dudley | Danny | [82,88,81,77] |
| Ellis | Evan | [91,93,87,93] |

**Transformation:**

First, you want to flag the students who did not take all four tests:

| Transformation Name | New formula |
|---------------------|-------------|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | IF(ARRAYLEN(Scores) < 4,"incomplete","") |
| Parameter: New column name | 'Error' |

This test flags Cameron Charles only.

The following transform sorts the array values in highest to lowest score:

| Transformation Name | Edit column with formula |
|---------------------|--------------------------|
| Parameter: Columns | Scores |
| Parameter: Formula | ARRAYSORT(Scores, 'descending') |

The following transforms extracts the first (highest) and last (lowest) value in each student's test scores, provided that they took four tests:

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | ARRAYELEMENTAT(Scores,0) |
| Parameter: New column name | 'highestScore' |

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | ARRAYELEMENTAT(Scores,3) |
| Parameter: New column name | 'lowestScore' |

> **Tip:** You could also generate the `Error` column when the `Scores4` column contains a null value. If no value exists in the array for the `ARRAYELEMENTAT` function, a null value is returned, which would indicate in this case an insufficient number of elements (test scores).

You can now track change in test scores:

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | SUBTRACT(highestScore,lowestScore) |
| Parameter: New column name | 'Score_range' |

**Results:**

| LastName | FirstName | Scores | Error | lowestScore | highestScore | Score_range |
|---|---|---|---|---|---|---|
| Allen | Amanda | [87,83,81,79] | | 79 | 87 | 8 |
| Bell | Bobby | [98,94,92,85] | | 85 | 98 | 13 |
| Charles | Cameron | [88,85,81] | incomplete | | 88 | |
| Dudley | Danny | [88,82,81,77] | | 77 | 88 | 11 |
| Ellis | Evan | [93,93,91,87] | | 87 | 93 | 6 |

# EXAMPLE - ARRAYSLICE and ARRAYMERGEELEMENTS Functions

This example illustrates how to generate an Array that is a slice of an another Array, based on index numbers. The elements of this Array can then be merged into a String value.

**Functions**:

| Item | Description |
|------|-------------|
| ARRAYSLICE Function | Returns an array containing a slice of the input array, as determined by starting and ending index parameters. |
| ARRAYMERGEELEMENTS Function | Merges the elements of an array in left to right order into a string. Values are optionally delimited by a provided delimiter. |

### Source:

The following set of arrays contain results, in order, of a series of races. From this list, the goal is to extract a list of the podium finishers for each race as a single string.

| RaceId | RaceResults |
|--------|-------------|
| 1 | ["racer3","racer5","racer2","racer1","racer6"] |
| 2 | ["racer6","racer4","racer2","racer1","racer3","racer5"] |
| 3 | ["racer4","racer3","racer5","racer2","racer6","racer1"] |
| 4 | ["racer1","racer2","racer3","racer5"] |
| 5 | ["racer5","racer2","racer4","racer6","racer3"] |

### Transformation:

From the list of arrays, the first step is to gather the top-3 finishers from each race:

| | |
|---|---|
| **Transformation Name** | New formula |
| **Parameter: Formula type** | Single row formula |
| **Parameter: Formula** | ARRAYSLICE(RaceResults, 0, 3) |
| **Parameter: New column name** | 'arrPodium' |

The above captures the first three values of the RaceResults arrays into a new set of arrays.

The next step is to merge this new set of arrays into a single string:

| | |
|---|---|
| **Transformation Name** | New formula |
| **Parameter: Formula type** | Single row formula |
| **Parameter: Formula** | ARRAYMERGEELEMENTS(arrPodium, ',') |
| **Parameter: New column name** | 'strPodium' |

**Results:**

| RaceId | RaceResults | arrPodium | strPodium |
|---|---|---|---|
| 1 | ["racer3","racer5","racer2","racer1","racer6"] | ["racer3","racer5","racer2"] | racer3,racer5,racer2 |
| 2 | ["racer6","racer4","racer2","racer1","racer3","racer5"] | ["racer6","racer4","racer2"] | racer6,racer4,racer2 |
| 3 | ["racer4","racer3","racer5","racer2","racer6","racer1"] | ["racer4","racer3","racer5"] | racer4,racer3,racer5 |
| 4 | ["racer1","racer2","racer3","racer5"] | ["racer1","racer2","racer3"] | racer1,racer2,racer3 |
| 5 | ["racer5","racer2","racer4","racer6","racer3"] | ["racer5","racer2","racer4"] | racer5,racer2,racer4 |

# EXAMPLE - ARRAYSTOMAP Function

This example illustrates how to use the ARRAYSTOMAP and KEYS functions to convert values in Array or Object data type of key-value pairs.

**Functions:**

| Item | Description |
|------|-------------|
| ARRAYSTOMAP Function | Combines one array containing keys and another array containing values into an Object of key-value pairs. |
| KEYS Function | Extracts the key values from an Object data type column and stores them in an array of String values. |

**Source:**

Your dataset contains master product data with product properties stored in two arrays of keys and values.

| ProdId | ProdCategory | ProdName | ProdKeys | ProdProperties |
|--------|-------------|----------|----------|----------------|
| S001 | Shirts | Crew Neck T-Shirt | ["type", "color", "fabric", "sizes"] | ["crew","blue","cotton","S,M,L","in stock","padded"] |
| S002 | Shirts | V-Neck T-Shirt | ["type", "color", "fabric", "sizes"] | ["v-neck","white","blend","S,M,L,XL","in stock","discount - seasonal"] |
| S003 | Shirts | Tanktop | ["type", "color", "fabric", "sizes"] | ["tank","red","mesh","XS,S,M","discount - clearance","in stock"] |
| S004 | Shirts | Turtleneck | ["type", "color", "fabric", "sizes"] | ["turtle","black","cotton","M,L,XL","out of stock","padded"] |

**Transformation:**

When the above data is loaded into the Transformer page, you might need to clean up the two array columns.

Using the following transform, you can map the first element of the first array as a key for the first element of the second, which is its value. You might notice that the number of keys and the number of values are not consistent. For the extra elements in the second array, the default key of `ProdMiscProperties` is used:

| Transformation Name | New formula |
|---------------------|-------------|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | ARRAYSTOMAP(ProdProperties, ProdValues, 'ProdMiscProperties') |
| Parameter: New column name | 'prodPropertyMap' |

You can now use the following steps to generate a new version of the keys:

| Transformation Name | Delete columns |
|---------------------|----------------|
| Parameter: Columns | ProdKeys |
| Parameter: Action | Delete selected columns |

| Transformation Name | New formula |
|---------------------|-------------|

| Parameter: Formula type | Single row formula |
|---|---|
| Parameter: Formula | KEYS(prodPropertyMap) |
| Parameter: New column name | 'ProdKeys' |

**Results:**

| ProdId | ProdCategory | ProdName | ProdKeys | ProdProperties | prodPropertyMap |
|---|---|---|---|---|---|
| S001 | Shirts | Crew Neck T-Shirt | ["type", "color", "fabric", "sizes","ProdMiscProperties"] | ["crew","blue","cotton","S,M,L","in stock","padded"] | { "type": [ "crew" ], "color": [ "blue" ], "fabric": [ "cotton" ], "sizes": [ "S,M,L" ], "ProdMiscProperties": [ "in stock", "padded" ] } |
| S002 | Shirts | V-Neck T-Shirt | ["type", "color", "fabric", "sizes","ProdMiscProperties"] | ["v-neck","white","blend","S,M,L,XL","in stock","discount - seasonal"] | { "type": [ "v-neck" ], "color": [ "white" ], "fabric": [ "blend" ], "sizes": [ "S,M,L,XL" ], "ProdMiscProperties": [ "in stock", "discount - seasonal" ] } |
| S003 | Shirts | Tanktop | ["type", "color", "fabric", "sizes","ProdMiscProperties"] | ["tank","red","mesh","XS,S,M","discount - clearance","in stock"] | { "type": [ "tank" ], "color": [ "red" ], "fabric": [ "mesh" ], "sizes": [ "XS,S,M" ], "ProdMiscProperties": [ "discount - clearance", "in stock" ] } |
| S004 | Shirts | Turtleneck | ["type", "color", "fabric", "sizes","ProdMiscProperties"] | ["turtle","black","cotton","M,L,XL","out of stock","padded"] | { "type": [ "turtle" ], "color": [ "black" ], "fabric": [ "cotton" ], "sizes": [ "M,L,XL" ], "ProdMiscProperties": [ "out of stock", "padded" ] } |

# EXAMPLE - Base64 Encoding Functions

This example demonstrates how to convert an input string to a base64-encoded value and back to ASCII text strings.

**Functions**:

| Item | Description |
|---|---|
| BASE64ENCODE Function | Converts an input value to base64 encoding with optional padding with an equals sign (=). Input can be of any type. Output type is String. |
| BASE64DECODE Function | Converts an input base64 value to text. Output type is String. |

**Source:**

The following example contains three columns of different data types:

| IntegerField | StringField | ssn |
|---|---|---|
| -2082863942 | This is a test string. | 987654321 |
| 2012994989 | "Hello, world." | 987654322 |
| -1637187918 | "Hello, world. Hello, world. Hello, world." | 987654323 |
| -1144194035 | fyi | 987654324 |
| -971872543 | | 987654325 |
| 353977583 | This is a test string. | 987-65-4321 |
| -366583667 | "Hello, world." | 987-65-4322 |
| -573117553 | "Hello, world. Hello, world. Hello, world." | 987-65-4323 |
| 2051041970 | fyi | 987-65-4324 |
| 522691086 | | 987-65-4325 |

**Transformation - encode:**

You can use the following transformation to encode all of the columns in your dataset:

| | |
|---|---|
| **Transformation Name** | `Edit column with formula` |
| **Parameter: Columns** | `All` |
| **Parameter: Formula** | `base64encode($col, true)` |

**Results - encode:**

The transformed dataset now looks like the following. Note the padding (equals signs) at the end of some of the values. Padding is added by default.

| IntegerField | StringField | ssn |
|---|---|---|
| LTIwODI4NjM5NDI= | VGhpcyBpcyBhIHRlc3Qgc3RyaW5nLg== | OTg3NjU0MzIx |
| MjAxMjk5NDk4OQ== | IkhlbGxvLCB3b3JsZC4i | OTg3NjU0MzIy |
| LTE2MzcxODc5MTg= | IkhlbGxvLCB3b3JsZC4gSGVsbG8sIHdvcmxkLiBIZWxsbywgd29ybGQuIg== | OTg3NjU0MzIz |
| LTExNDQxOTQwMzU= | Znlp | OTg3NjU0MzI0 |

| | | |
|---|---|---|
| LTk3MTg3MjU0Mw== | | OTg3NjU0MzI1 |
| MzUzOTc3NTgz | VGhpcyBpcyBhIHRlc3Qgc3RyaW5nLg== | OTg3LTY1LTQzMjE= |
| LTM2NjU4MzY2Nw== | IkhlbGxvLCB3b3JsZC4i | OTg3LTY1LTQzMjI= |
| LTU3MzExNzU1Mw== | IkhlbGxvLCB3b3JsZC4gSGVsbG8sIHdvcmxkLiBIZWxsbywgd29ybGQuIg== | OTg3LTY1LTQzMjM= |
| MjA1MTA0MTk3MA== | Znlp | OTg3LTY1LTQzMjQ= |
| NTIyNjkxMDg2 | | OTg3LTY1LTQzMjU= |

### Transformation - decode:

The following transformation can be used to decode all of the columns:

| | |
|---|---|
| **Transformation Name** | `Edit column with formula` |
| **Parameter: Columns** | `All` |
| **Parameter: Formula** | `base64decode($col)` |

### Results - decode:

| IntegerField | StringField | ssn |
|---|---|---|
| -2082863942 | This is a test string. | 987654321 |
| 2012994989 | "Hello, world." | 987654322 |
| -1637187918 | "Hello, world. Hello, world. Hello, world." | 987654323 |
| -1144194035 | fyi | 987654324 |
| -971872543 | | 987654325 |
| 353977583 | This is a test string. | 987-65-4321 |
| -366583667 | "Hello, world." | 987-65-4322 |
| -573117553 | "Hello, world. Hello, world. Hello, world." | 987-65-4323 |
| 2051041970 | fyi | 987-65-4324 |
| 522691086 | | 987-65-4325 |

# EXAMPLE - Case Functions

This example demonstrates functions that can be used to change the case of String values.

**Functions:**

| Item | Description |
|---|---|
| UPPER Function | All alphabetical characters in the input value are converted to uppercase in the output value. |
| LOWER Function | All alphabetical characters in the input value are converted to lowercase in the output value. |
| PROPER Function | Converts an input string to propercase. Input can be a column reference or a string literal. |

**Source:**

In the following example, you can see a number of input values in the leftmost column. The output columns are blank.

| input | uppercase | lowercase | propercase |
|---|---|---|---|
| AbCdEfGh IjKlMnO | | | |
| go West, young man! | | | |
| Oh, *(*$%(&! That HURT! | | | |
| A11 w0rk and n0 0play makes Jack a dull boy. | | | |

**Transformation:**

To generate uppercase, lowercase, and propercase values in the output columns, use the following transforms:

| Transformation Name | Edit column with formula |
|---|---|
| **Parameter: Columns** | uppercase |
| **Parameter: Formula** | UPPER(input) |

| Transformation Name | Edit column with formula |
|---|---|
| **Parameter: Columns** | lowercase |
| **Parameter: Formula** | LOWER(input) |

| Transformation Name | Edit column with formula |
|---|---|
| **Parameter: Columns** | propercase |
| **Parameter: Formula** | PROPER(input) |

**Results:**

| input | uppercase | lowercase | propercase |
|---|---|---|---|
| AbCdEfGh IjKlMnO | ABCDEFGH IJKLMNO | abcdefgh ijklmno | Abcdefgh Ijklmno |
| go West, young man! | GO WEST, YOUNG MAN! | go west, young man! | Go West, Young Man! |

| | | | |
|---|---|---|---|
| Oh, *(*$%(&! That HURT! | OH, *(*$%(&! THAT HURT! | oh, *(*$%(&! that hurt! | Oh, *(*$%(&! That Hurt! |
| A11 w0rk and n0 0play makes Jack a dull boy. | A11 W0RK AND N0 0PLAY MAKES JACK A DULL BOY. | a11 w0rk and n0 0play makes jack a dull boy. | A11 W0rk And N0 0play Makes Jack A Dull Boy. |

# EXAMPLE - Comparison Functions1

This example illustrates the comparison functions in Designer Cloud Powered by Trifacta® Enterprise Edition.

**Functions**:

| Item | Description |
|---|---|
| LESSTHAN Function | Returns `true` if the first argument is less than but not equal to the second argument. Equivalent to the `<` operator. |
| LESSTHANEQUAL Function | Returns `true` if the first argument is less than or equal to the second argument. Equivalent to the `<=` operator. |
| EQUAL Function | Returns `true` if the first argument is equal to the second argument. Equivalent to the `=` operator. |
| NOTEQUAL Function | Returns `true` if the first argument is not equal to the second argument. Equivalent to the `<>` or `!=` operator. |
| GREATERTHAN Function | Returns `true` if the first argument is greater than but not equal to the second argument. Equivalent to the `>` operator. |
| GREATERTHANEQUAL Function | Returns `true` if the first argument is greater than or equal to the second argument. Equivalent to the `>=` operator. |

**Source:**

| colA | colB |
|---|---|
| 1 | 11 |
| 2 | 10 |
| 3 | 9 |
| 4 | 8 |
| 5 | 7 |
| 6 | 6 |
| 7 | 5 |
| 8 | 4 |
| 9 | 3 |
| 10 | 2 |
| 11 | 1 |

**Transformation:**

Add the following transforms to your recipe, one for each comparison function:

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | LESSTHAN(colA, colB) |
| Parameter: New column name | 'lt' |

| Transformation Name | New formula |
|---|---|

| Parameter: Formula type | Single row formula |
|---|---|
| Parameter: Formula | LESSTHANEQUAL(colA, colB) |
| Parameter: New column name | 'lte' |

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | EQUAL(colA, colB) |
| Parameter: New column name | 'eq' |

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | NOTEQUAL(colA, colB) |
| Parameter: New column name | 'neq' |

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | GREATERTHAN(colA, colB) |
| Parameter: New column name | 'gt' |

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | GREATERTHANEQUAL(colA, colB) |
| Parameter: New column name | 'gte' |

**Results:**

| colA | colB | gte | gt | neq | eq | lte | lt |
|---|---|---|---|---|---|---|---|
| 1 | 11 | false | false | true | false | true | true |
| 2 | 10 | false | false | true | false | true | true |
| 3 | 9 | false | false | true | false | true | true |
| 4 | 8 | false | false | true | false | true | true |
| 5 | 7 | false | false | true | false | true | true |
| 6 | 6 | true | false | false | true | true | false |
| 7 | 5 | true | true | true | false | false | false |
| 8 | 4 | true | true | true | false | false | false |
| 9 | 3 | true | true | true | false | false | false |

| 10 | 2 | true | true | true | false | false | false |
| 11 | 1 | true | true | true | false | false | false |

# EXAMPLE - Comparison Functions2

This example demonstrates functions for comparing the relative values of two functions.

**Functions:**

| Item | Description |
|------|-------------|
| LESSTHANEQUAL Function | Returns `true` if the first argument is less than or equal to the second argument. Equivalent to the `<=` operator. |
| LESSTHAN Function | Returns `true` if the first argument is less than but not equal to the second argument. Equivalent to the `<` operator. |
| GREATERTHANEQUAL Function | Returns `true` if the first argument is greater than or equal to the second argument. Equivalent to the `>=` operator. |
| GREATERTHAN Function | Returns `true` if the first argument is greater than but not equal to the second argument. Equivalent to the `>` operator. |

In the town of Circleville, citizens are allowed to maintain a single crop circle in their backyard, as long as it confirms to the town regulations. Below is some data on the size of crop circles in town, with a separate entry for each home. Limits are displayed in the adjacent columns, with the `inclusive` columns indicating whether the minimum or maximum values are inclusive.

> **Tip:** As part of this exercise, you can see how to you can extend your recipe to perform some simple financial analysis of the data.

**Source:**

| Location | Radius_ft | minRadius_ft | minInclusive | maxRadius_ft | maxInclusive |
|----------|-----------|--------------|--------------|--------------|--------------|
| House1 | 55.5 | 10 | Y | 25 | N |
| House2 | 12 | 10 | Y | 25 | N |
| House3 | 14.25 | 10 | Y | 25 | N |
| House4 | 3.5 | 10 | Y | 25 | N |
| House5 | 27 | 10 | Y | 25 | N |

**Transformation:**

After the data is loaded into the Transformer page, you can begin comparing column values:

| Transformation Name | `New formula` |
|---------------------|---------------|
| **Parameter: Formula type** | `Single row formula` |
| **Parameter: Formula** | `LESSTHANEQUAL(Radius_ft,minRadius_ft)` |
| **Parameter: New column name** | `'tooSmall'` |

While accurate, the above transform does not account for the `minInclusive` value, which may be changed as part of your steps. Instead, you can delete the previous transform and use the following, which factors in the other column:

| Transformation | `New formula` |
|----------------|---------------|

| Name | |
|---|---|
| **Parameter: Formula type** | `Single row formula` |
| **Parameter: Formula** | `IF(minInclusive == 'Y',LESSTHANEQUAL(Radius_ft,minRadius_ft),` `LESSTHAN(Radius_ft,minRadius_ft))` |
| **Parameter: New column name** | `'tooSmall'` |

In this case, the `IF` function tests whether the minimum value is inclusive (values of `10` are allowed). If so, the `LESSTHANEQUAL` function is applied. Otherwise, the `LESSTHAN` function is applied. For the maximum limit, the following step applies:

| **Transformation Name** | `New formula` |
|---|---|
| **Parameter: Formula type** | `Single row formula` |
| **Parameter: Formula** | `IF(maxInclusive == 'Y', GREATERTHANEQUAL(Radius_ft,` `maxRadius_ft),GREATERTHAN(Radius_ft,maxRadius_ft))` |
| **Parameter: New column name** | `'tooBig'` |

Now, you can do some analysis of this data. First, you can insert a column containing the amount of the fine per foot above the maximum or below the minimum. Before the first `derive` command, insert the following, which is the fine (`$15.00`) for each foot above or below the limits:

| **Transformation Name** | `New formula` |
|---|---|
| **Parameter: Formula type** | `Single row formula` |
| **Parameter: Formula** | `15` |
| **Parameter: New column name** | `'fineDollarsPerFt'` |

At the end of the recipe, add the following new line, which calculates the fine for crop circles that are too small:

| **Transformation Name** | `New formula` |
|---|---|
| **Parameter: Formula type** | `Single row formula` |
| **Parameter: Formula** | `IF(tooSmall == 'true', (minRadius_ft - Radius_ft) *` `fineDollarsPerFt, 0.0)` |
| **Parameter: New column name** | `'fine_Dollars'` |

The above captures the too-small violations. To also capture the too-big violations, change the above to the following:

| **Transformation Name** | `New formula` |
|---|---|
| **Parameter: Formula type** | `Single row formula` |
| | |

| Parameter: Formula | IF(tooSmall == 'true', (minRadius_ft - Radius_ft) * fineDollarsPerFt, if(tooBig == 'true', (Radius_ft - maxRadius_ft) * fineDollarsPerFt, '0.0')) |
|---|---|
| Parameter: New column name | 'fine_Dollars' |

In place of the original "false" expression (0.0), the above adds the test for the too-big values, so that all fines are included in a single column. You can reformat the fine_Dollars column to be in dollar format:

| Transformation Name | Edit column with formula |
|---|---|
| Parameter: Columns | fine_Dollars |
| Parameter: Formula | NUMFORMAT(fine_Dollars, '$###.00') |

**Results:**

After you delete the columns used in the calculation and move the remaining ones, you should end up with a dataset similar to the following:

| Location | fineDollarsPerFt | Radius_ft | minRadius_ft | minInclusive | maxRadius_ft | maxInclusive | fineDollars |
|---|---|---|---|---|---|---|---|
| House1 | 15 | 55.5 | 10 | Y | 25 | N | $457.50 |
| House2 | 15 | 12 | 10 | Y | 25 | N | $0.00 |
| House3 | 15 | 14.25 | 10 | Y | 25 | N | $0.00 |
| House4 | 15 | 3.5 | 10 | Y | 25 | N | $97.50 |
| House5 | 15 | 27 | 10 | Y | 25 | N | $30.00 |

Now that you have created all of the computations for generating these values, you can change values for minRadius_ft, maxRadius_ft, and fineDollarsPerFt to analyze the resulting fine revenue. Before or after the transform where you set the value for fineDollarsPerFt, you can insert something like the following:

| Transformation Name | Edit column with formula |
|---|---|
| Parameter: Columns | minRadius_ft |
| Parameter: Formula | '12.5' |

After the step is added, select the last line in the recipe. Then, you can see how the values in the fineDollars column have been updated.

# EXAMPLE - Comparison Functions Equal

This example demonstrates comparison functions.

**Functions:**

| Item | Description |
|------|-------------|
| EQUAL Function | Returns `true` if the first argument is equal to the second argument. Equivalent to the = operator. |
| NOTEQUAL Function | Returns `true` if the first argument is not equal to the second argument. Equivalent to the `<>` or `!=` operator. |
| ISEVEN Function | Returns `true` if the argument is an even value. Argument can be an Integer, a function returning Integers, or a column reference. |
| ISODD Function | Returns `true` if the argument is an odd value. Argument can be an Integer, a function returning Integers, or a column reference. |

In this example, the dataset contains current measurements of the sides of rectangular areas next to the size of those areas as previously reported. Using these functions, you can perform some light analysis of the data.

**Source:**

| sideA | sideB | reportedArea |
|-------|-------|--------------|
| 4 | 14 | 56 |
| 6 | 6 | 35 |
| 8 | 4 | 32 |
| 15 | 15 | 200 |
| 4 | 7 | 28 |
| 12 | 6 | 70 |
| 9 | 9 | 81 |

**Transformation:**

In the first test, you are determining if the four-sided area is a square, based on a comparison of the measured values for `sideA` and `sideB`:

| Transformation Name | New formula |
|---------------------|-------------|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | EQUAL(sideA, sideB) |
| Parameter: New column name | 'isSquare' |

Next, you can use the reported sides to calculate the area of the shape and compare it to the area previously reported:

| Transformation Name | New formula |
|---------------------|-------------|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | NOTEQUAL(sideA * sideB, reportedArea) |

| Parameter: New column name | `'isValidData'` |
| --- | --- |

You can also compute if the reportedArea can be divided into even square units:

| Transformation Name | `New formula` |
| --- | --- |
| Parameter: Formula type | `Single row formula` |
| Parameter: Formula | `ISEVEN(reportedArea)` |
| Parameter: New column name | `'isReportedAreaEven'` |

You can test if either measured side is an odd number of units:

| Transformation Name | `New formula` |
| --- | --- |
| Parameter: Formula type | `Single row formula` |
| Parameter: Formula | `IF((ISODD(sideA) == true) OR (ISODD(sideB) == true),TRUE, FALSE)` |
| Parameter: New column name | `'isSideOdd'` |

**Results:**

| sideA | sideB | reportedArea | isSquare | isValidData | isReportedAreaEven | isSideOdd |
| --- | --- | --- | --- | --- | --- | --- |
| 4 | 14 | 56 | FALSE | FALSE | TRUE | FALSE |
| 6 | 6 | 35 | TRUE | TRUE | TRUE | FALSE |
| 8 | 4 | 32 | FALSE | FALSE | TRUE | FALSE |
| 15 | 15 | 200 | TRUE | TRUE | TRUE | TRUE |
| 4 | 7 | 28 | FALSE | FALSE | TRUE | TRUE |
| 12 | 6 | 70 | FALSE | TRUE | TRUE | FALSE |
| 9 | 9 | 81 | TRUE | FALSE | FALSE | FALSE |

# EXAMPLE - Conditional Calculations Functions

This example illustrates how to use the conditional calculation functions.

**Functions**:

| Item | Description |
|------|-------------|
| AVERAGEIF Function | Generates the average value of rows in each group that meet a specific condition. Generated value is of Decimal type. |
| MINIF Function | Generates the minimum value of rows in each group that meet a specific condition. Inputs can be Integer, Decimal, or Datetime. |
| MAXIF Function | Generates the maximum value of rows in each group that meet a specific condition. Inputs can be Integer, Decimal, or Datetime. |
| VARIF Function | Generates the variance of values by group in a column that meet a specific condition. |
| STDEVIF Function | Generates the standard deviation of values by group in a column that meet a specific condition. |

**Source:**

Here is some example weather data:

| date | city | rain | temp | wind |
|------|------|------|------|------|
| 1/23/17 | Valleyville | 0.00 | 12.8 | 6.7 |
| 1/23/17 | Center Town | 0.31 | 9.4 | 5.3 |
| 1/23/17 | Magic Mountain | 0.00 | 0.0 | 7.3 |
| 1/24/17 | Valleyville | 0.25 | 17.2 | 3.3 |
| 1/24/17 | Center Town | 0.54 | 1.1 | 7.6 |
| 1/24/17 | Magic Mountain | 0.32 | 5.0 | 8.8 |
| 1/25/17 | Valleyville | 0.02 | 3.3 | 6.8 |
| 1/25/17 | Center Town | 0.83 | 3.3 | 5.1 |
| 1/25/17 | Magic Mountain | 0.59 | -1.7 | 6.4 |
| 1/26/17 | Valleyville | 1.08 | 15.0 | 4.2 |
| 1/26/17 | Center Town | 0.96 | 6.1 | 7.6 |
| 1/26/17 | Magic Mountain | 0.77 | -3.9 | 3.0 |
| 1/27/17 | Valleyville | 1.00 | 7.2 | 2.8 |
| 1/27/17 | Center Town | 1.32 | 20.0 | 0.2 |
| 1/27/17 | Magic Mountain | 0.77 | 5.6 | 5.2 |
| 1/28/17 | Valleyville | 0.12 | -6.1 | 5.1 |
| 1/28/17 | Center Town | 0.14 | 5.0 | 4.9 |
| 1/28/17 | Magic Mountain | 1.50 | 1.1 | 0.4 |
| 1/29/17 | Valleyville | 0.36 | 13.3 | 7.3 |
| 1/29/17 | Center Town | 0.75 | 6.1 | 9.0 |
| 1/29/17 | Magic Mountain | 0.60 | 3.3 | 6.0 |

**Transformation:**

The following computes average temperature for rainy days by city:

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | AVERAGEIF(temp, rain > 0) |
| Parameter: Group rows by | city |
| Parameter: New column name | 'avgTempWRain' |

The following computes maximum wind for sub-zero days by city:

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | MAXIF(wind,temp < 0) |
| Parameter: Group rows by | city |
| Parameter: New column name | 'maxWindSubZero' |

This step calculates the minimum temp when the wind is less than 5 mph by city:

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | MINIF(temp,wind<5) |
| Parameter: Group rows by | city |
| Parameter: New column name | 'minTempWind5' |

This step computes the variance in temperature for rainy days by city:

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | VARIF(temp,rain >0) |
| Parameter: Group rows by | city |
| Parameter: New column name | 'varTempWRain' |

The following computes the standard deviation in rainfall for Center Town:

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | STDEVIF(rain,city=='Center Town') |
| Parameter: Group rows by | city |

| | |
|---|---|
| **Parameter: New column name** | `'stDevRainCT'` |

You can use the following transforms to format the generated output. Note the `$col` placeholder value for the multi-column transforms:

| | |
|---|---|
| **Transformation Name** | `Edit column with formula` |
| **Parameter: Columns** | `stDevRainCenterTown,maxWindSubZero` |
| **Parameter: Formula** | `numformat($col,'##.##')` |

Since the following rely on data that has only one significant digit, you should format them differently:

| | |
|---|---|
| **Transformation Name** | `Edit column with formula` |
| **Parameter: Columns** | `varTempWRain,avgTempWRain,minTempWind5` |
| **Parameter: Formula** | `numformat($col,'##.#')` |

**Results:**

| date | city | rain | temp | wind | avgTempWRain | maxWindSubZero | minTempWind5 | varTempWRain | stDevRain |
|---|---|---|---|---|---|---|---|---|---|
| 1/23 /17 | Valley ville | 0.00 | 12.8 | 6.7 | 8.3 | 5.1 | 7.2 | 63.8 | 0.37 |
| 1/23 /17 | Cente r Town | 0.31 | 9.4 | 5.3 | 7.3 | | 5 | 32.6 | 0.37 |
| 1/23 /17 | Magic Mount ain | 0.00 | 0.0 | 7.3 | 1.6 | 6.43 | -3.9 | 12 | 0.37 |
| 1/24 /17 | Valley ville | 0.25 | 17.2 | 3.3 | 8.3 | 5.1 | 7.2 | 63.8 | 0.37 |
| 1/24 /17 | Cente r Town | 0.54 | 1.1 | 7.6 | 7.3 | | 5 | 32.6 | 0.37 |
| 1/24 /17 | Magic Mount ain | 0.32 | 5.0 | 8.8 | 1.6 | 6.43 | -3.9 | 12 | 0.37 |
| 1/25 /17 | Valley ville | 0.02 | 3.3 | 6.8 | 8.3 | 5.1 | 7.2 | 63.8 | 0.37 |
| 1/25 /17 | Cente r Town | 0.83 | 3.3 | 5.1 | 7.3 | | 5 | 32.6 | 0.37 |
| 1/25 /17 | Magic Mount ain | 0.59 | -1.7 | 6.4 | 1.6 | 6.43 | -3.9 | 12 | 0.37 |
| 1/26 /17 | Valley ville | 1.08 | 15.0 | 4.2 | 8.3 | 5.1 | 7.2 | 63.8 | 0.37 |
| 1/26 /17 | Cente r Town | 0.96 | 6.1 | 7.6 | 7.3 | | 5 | 32.6 | 0.37 |
| 1/26 /17 | Magic Mount ain | 0.77 | -3.9 | 3.0 | 1.6 | 6.43 | -3.9 | 12 | 0.37 |
| 1/27 /17 | Valley ville | 1.00 | 7.2 | 2.8 | 8.3 | 5.1 | 7.2 | 63.8 | 0.37 |

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 1/27 /17 | Center Town | 1.32 | 20.0 | 0.2 | 7.3 | | 5 | 32.6 | 0.37 |
| 1/27 /17 | Magic Mountain | 0.77 | 5.6 | 5.2 | 1.6 | 6.43 | -3.9 | 12 | 0.37 |
| 1/28 /17 | Valley ville | 0.12 | -6.1 | 5.1 | 8.3 | 5.1 | 7.2 | 63.8 | 0.37 |
| 1/28 /17 | Center Town | 0.14 | 5.0 | 4.9 | 7.3 | | 5 | 32.6 | 0.37 |
| 1/28 /17 | Magic Mountain | 1.50 | 1.1 | 0.4 | 1.6 | 6.43 | -3.9 | 12 | 0.37 |
| 1/29 /17 | Valley ville | 0.36 | 13.3 | 7.3 | 8.3 | 5.1 | 7.2 | 63.8 | 0.37 |
| 1/29 /17 | Center Town | 0.75 | 6.1 | 9.0 | 7.3 | | 5 | 32.6 | 0.37 |
| 1/29 /17 | Magic Mountain | 0.60 | 3.3 | 6.0 | 1.6 | 6.43 | -3.9 | 12 | 0.37 |

# EXAMPLE - COUNT Functions

This example demonstrates how to count the number of values and non-null values within a group.

**Functions**:

| Item | Description |
|---|---|
| COUNTA Function | Generates the count of non-null rows in a specified column, optionally counted by group. Generated value is of Integer type. |
| COUNTDISTINCT Function | Generates the count of distinct values in a specified column, optionally counted by group. Generated value is of Integer type. |

**Source:**

In the following example, the seventh row is an empty string, and the eighth row is a null value.

| rowId | Val |
|---|---|
| r001 | val1 |
| r002 | val1 |
| r003 | val1 |
| r004 | val2 |
| r005 | val2 |
| r006 | val3 |
| r007 | (empty) |
| r008 | (null) |

**Transformation:**

Apply a `COUNTA` function on the source column:

| Transformation Name | New formula |
|---|---|
| **Parameter: Formula type** | Single row formula |
| **Parameter: Formula** | COUNTA(Val) |
| **Parameter: New column name** | 'fctnCounta' |

Apply a `COUNTDISTINCT` function on the source:

| Transformation Name | New formula |
|---|---|
| **Parameter: Formula type** | Single row formula |
| **Parameter: Formula** | COUNTDISTINCT(Val) |
| **Parameter: New column name** | 'fctnCountdistinct' |

**Results:**

Below, both functions count the number of values in the column, with `COUNTDISTINCT` counting distinct values only. The empty value for `r007` is counted by both functions.

| rowId | Val | fctnCountdistinct | fctnCounta |
|---|---|---|---|
| r001 | val1 | 4 | 7 |
| r002 | val1 | 4 | 7 |
| r003 | val1 | 4 | 7 |
| r004 | val2 | 4 | 7 |
| r005 | val2 | 4 | 7 |
| r006 | val3 | 4 | 7 |
| r007 | (empty) | 4 | 7 |
| r008 | (null) | 4 | 7 |

# EXAMPLE - COUNTIF Functions

This example demonstrates how to count the number of values within a group, based on a specified conditional test.

**Functions**:

| Item | Description |
|------|-------------|
| COUNTIF Function | Generates the count of rows in each group that meet a specific condition. Generated value is of Integer type. |
| COUNTAIF Function | Generates the count of non-null values for rows in each group that meet a specific condition. |

**Source:**

The following data identifies sales figures by salespeople for a week:

| EmployeeId | Date | Sales |
|------------|------|-------|
| S001 | 1/23/17 | 25 |
| S002 | 1/23/17 | 40 |
| S003 | 1/23/17 | 48 |
| S001 | 1/24/17 | 81 |
| S002 | 1/24/17 | 11 |
| S003 | 1/24/17 | 25 |
| S001 | 1/25/17 | 9 |
| S002 | 1/25/17 | 40 |
| S003 | 1/25/17 |  |
| S001 | 1/26/17 | 77 |
| S002 | 1/26/17 | 83 |
| S003 | 1/26/17 |  |
| S001 | 1/27/17 | 17 |
| S002 | 1/27/17 | 71 |
| S003 | 1/27/17 | 29 |
| S001 | 1/28/17 |  |
| S002 | 1/28/17 |  |
| S003 | 1/28/17 | 14 |
| S001 | 1/29/17 | 2 |
| S002 | 1/29/17 | 7 |
| S003 | 1/29/17 | 99 |

**Transformation:**

You are interested in the count of dates during the week when each salesperson sold less than 50 units, not factoring the weekend. First, you try the following:

| Transformation Name | Pivot columns |
|---|---|
| Parameter: Row labels | EmployeeId |
| Parameter: Values | COUNTIF(Sales < 50) |
| Parameter: Max columns to create | 1 |

You notice, however, that the blank values, when employees were sick or had vacation, are being counted. Additionally, this step does not filter out the weekend. You must identify the weekend days using the WEEKDAY function:

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | WEEKDAY(Date) |
| Parameter: New column name | 'DayOfWeek' |

If DayOfWeek > 5, then it is a weekend date. For further precision, you can use the COUNTAIF function to remove the nulls:

| Transformation Name | Pivot columns |
|---|---|
| Parameter: Row labels | EmployeeId |
| Parameter: Values | COUNTAIF(Sales, DayOfWeek<6) |
| Parameter: Max columns to create | 1 |

The above counts the non-null values in Sales when the day of the week is not a weekend day, as grouped by individual employee.

**Results:**

| EmployeeId | countaif_Sales |
|---|---|
| S001 | 5 |
| S002 | 4 |
| S003 | 4 |

# EXAMPLE - Countpattern Transform

This example demonstrates how to count the number of occurrences of text patterns in a column.

**Functions:**

| Item | Description |
|------|-------------|
| IF Function | The `IF` function allows you to build if/then/else conditional logic within your transforms. |

**Source:**

The dataset below contains fictitious tweet information shortly after the release of an application called, "Myco ExampleApp".

| Date | twitterId | isEmployee | tweet |
|------|-----------|------------|-------|
| 11/5/15 | lawrencetlu38141 | FALSE | Just downloaded Myco ExampleApp! Transforming data in 5 mins! |
| 11/5/15 | petramktng024 | TRUE | Try Myco ExampleApp, our new free data wrangling app! See www.example.com. |
| 11/5/15 | joetri221 | TRUE | Proud to announce the release of Myco ExampleApp, the free version of our enterprise product. Check it out at www.example.com. |
| 11/5/15 | datadaemon994 | FALSE | Great start with Myco ExampleApp. Super easy to use, and actually fun. |
| 11/5/15 | 99redballoons99 | FALSE | Liking this new ExampleApp! Good job, guys! |
| 11/5/15 | bigdatadan7182 | FALSE | @support, how can I find example datasets for use with your product? |

There are two areas of analysis:

- For non-employees, you want to know if they are mentioning the new product by name.
- For employees, you want to know if they are including cross-references to the web site as part of their tweet.

**Transformation:**

The following counts the occurrences of the string `ExampleApp` in the `tweet` column. Note the use of the `ignoreCase` parameter to capture capitalization differences:

| Transformation Name | Count matches |
|---------------------|---------------|
| **Parameter: Column** | `tweet` |
| **Parameter: Option** | `Text or pattern` |
| **Parameter: Text or pattern to count** | `'ExampleApp'` |
| **Parameter: Ignore case** | `true` |

For non-employees, you want to track if they have mentioned the product in their tweet:

| Transformation Name | New formula |
|---------------------|-------------|
| **Parameter: Formula type** | `Single row formula` |

| Parameter: Formula | `if(isEmployee=='FALSE' && countpattern_tweet=='1',true, false)` |
|---|---|
| Parameter: New column name | `'nonEmployeeExampleAppMentions'` |

The following counts the occurrences of `example.com` in their tweets:

| Transformation Name | `Count matches` |
|---|---|
| Parameter: Column | `tweet` |
| Parameter: Option | `Text or pattern` |
| Parameter: Text or pattern to count | `'example.com'` |
| Parameter: Ignore case | `true` |

For employees, you want to track if they included the above cross-reference in their tweets:

| Transformation Name | `New formula` |
|---|---|
| Parameter: Formula type | `Single row formula` |
| Parameter: Formula | `if(isEmployee=='TRUE' && countpattern_tweet1 == 1, true, false)` |
| Parameter: New column name | `'employeeWebsiteCrossRefs'` |

### Results:

After you delete the two columns tabulating the counts, you end up with the following:

| Date | twitterId | isEmployee | tweet | employeeWebsiteCrossRefs | nonEmployeeExampleAppMentions |
|---|---|---|---|---|---|
| 11/5 /15 | lawrencetl u38141 | FALSE | Just downloaded Myco ExampleApp! Transforming data in 5 mins! | false | true |
| 11/5 /15 | petramktn g024 | TRUE | Try Myco ExampleApp, our new free data wrangling app! See www. example.com. | true | false |
| 11/5 /15 | joetri221 | TRUE | Proud to announce the release of Myco ExampleApp, the free version of our enterprise product. Check it out at www. example.com. | true | false |
| 11/5 /15 | datadaem on994 | FALSE | Great start with Myco ExampleApp. Super easy to use, and actually fun. | false | true |
| 11/5 /15 | 99redballo ons99 | FALSE | Liking this new ExampleApp! Good job, guys! | false | true |
| 11/5 /15 | bigdatada n7182 | FALSE | @support, how can I find example datasets | false | false |

| | | | for use with your product? | | |
|---|---|---|---|---|---|

# EXAMPLE - DATE and TIME Functions

This example provides an overview on various date and time functions.

**Functions**:

| Item | Description |
|------|-------------|
| DATE Function | Generates a date value from three inputs of Integer type: year, month, and day. |
| TIME Function | Generates time values from three inputs of Integer type: hour, minute, and second. |
| DATETIME Function | Generates a Datetime value from the following inputs of Integer type: year, month, day, hour, minute, and second. |
| DATEFORMAT Function | Formats a specified Datetime set of values according to the specified date format. Source values can be a reference to a column containing Datetime values. |

**Source:**

| year | month | day | hour | minute | second |
|------|-------|-----|------|--------|--------|
| 2016 | 10 | 11 | 2 | 3 | 0 |
| 2015 | 11 | 20 | 15 | 22 | 30 |
| 2014 | 12 | 25 | 18 | 30 | 45 |

**Transformation:**

| Transformation Name | New formula |
|---------------------|-------------|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | DATEFORMAT(DATE (year, month, day),'yyyy-MM-dd') |
| Parameter: New column name | 'fctn_date' |

| Transformation Name | New formula |
|---------------------|-------------|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | DATEFORMAT(TIME (hour, minute, second),'HH-mm-ss') |
| Parameter: New column name | 'fctn_time' |

| Transformation Name | New formula |
|---------------------|-------------|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | DATEFORMAT(DATETIME (year, month, day, hour, minute, second),'yyyy-MM-dd-HH:mm:ss') |
| Parameter: New column name | 'fctn_datetime' |

**Results:**

> **NOTE:** All inputs must be inferred as Integer type and must be valid values for the specified input. For example, month values must be integers between 1 and 12, inclusive.

| year | month | day | hour | minute | second | fctn_date | fctn_time | fctn_datetime |
|------|-------|-----|------|--------|--------|-----------|-----------|---------------|
| 2016 | 10 | 11 | 2 | 3 | 0 | 2016-10-11 | 02-03-00 | 2016-10-11-02:03:00 |
| 2015 | 11 | 20 | 15 | 22 | 30 | 2015-11-20 | 15-22-30 | 2015-11-20-15:22:30 |
| 2014 | 12 | 25 | 18 | 30 | 45 | 2014-12-25 | 18-30-45 | 2014-12-25-18:30:45 |

You can apply other date and time functions to the generated columns. For an example, see *YEAR Function*.

# EXAMPLE - Date Difference Functions

This example demonstrates how to calculate the number of days between two input dates.

**Functions**:

| Item | Description |
|------|-------------|
| DATEDIF Function | Calculates the difference between two valid date values for the specified units of measure. |
| NETWORKDAYS Function | Calculates the number of working days between two specified dates, assuming Monday - Friday workweek. Optional list of holidays can be specified. |
| NETWORKDAYS INTL Function | Calculates the number of working days between two specified dates. Optionally, you can specify which days of the week are working days as an input parameter. Optional list of holidays can be specified. |
| WORKDAY Function | Calculates the work date that is before or after a start date, as specified by a number of days. A set of holiday dates can be optionally specified. |
| WORKDAYINTL Function | Calculates the work date that is before or after a start date, as specified by a number of days. You can also specify which days of the week are working days and a list of holidays via parameters. |

**Source:**

The following dataset contains two columns of dates.

- The first column values are constant. This date falls on a Monday.

| Date1 | Date2 |
|-------|-------|
| 2020-03-09 | 2020-03-13 |
| 2020-03-09 | 2020-03-06 |
| 2020-03-09 | 2020-03-16 |
| 2020-03-09 | 2020-03-23 |
| 2020-03-09 | 2020-04-10 |
| 2020-03-09 | 2021-03-10 |

**Transformation:**

The first transformation calculates the number of raw days between the two dates:

| Transformation Name | New formula |
|---------------------|-------------|
| **Parameter: Formula type** | Single row formula |
| **Parameter: Formula** | datedif(Date1, Date2, day) |
| **Parameter: New column name** | 'datedif' |

This step computes the number of working days between the two dates. Assumptions:

- Workweek is Monday - Friday.
- There are no holidays.

| Transformation Name | New formula |
|---------------------|-------------|
| **Parameter: Formula type** | Single row formula |

| Parameter: Formula | `networkdays(Date1, Date2, [])` |
|---|---|
| Parameter: New column name | `'networkDays'` |

For some, March 17 is an important date, especially if you are Irish. To add St. Patrick's Day to the list of holidays, you could add the following transformation:

| Transformation Name | `New formula` |
|---|---|
| Parameter: Formula type | `Single row formula` |
| Parameter: Formula | `networkdays(Date1, Date2, ['2020-03-17'])` |
| Parameter: New column name | `'networkDaysStPatricks'` |

In the following transformation, the NETWORKDAYSINTL function is applied so that you can specify the working days in the week:

| Transformation Name | `New formula` |
|---|---|
| Parameter: Formula type | `Single row formula` |
| Parameter: Formula | `networkdaysintl(Date1, Date2, '1000011', [])` |
| Parameter: New column name | `'networkDaysIntl'` |

The following two functions enable you to calculate a specific working date based on an input date and integer number of days before or after it. In the following, the date that is five working days before the `Date2` column is computed:

| Transformation Name | `New formula` |
|---|---|
| Parameter: Formula type | `Single row formula` |
| Parameter: Formula | `workday(Date2, -5)` |
| Parameter: New column name | `'workday'` |

Suppose you wish to factor in a four-day workweek, in which Friday through Sunday is considered the weekend:

| Transformation Name | `New formula` |
|---|---|
| Parameter: Formula type | `Single row formula` |
| Parameter: Formula | `workdayintl(Date2, -5,'0000111')` |
| Parameter: New column name | `'workdayintl'` |

**Results:**

| Date1 | Date2 | workdayintl | workday | networkDaysIntl | networkDaysStPatricks | networkDays | datedif |
|---|---|---|---|---|---|---|---|
| 2020-03-09 | 2020-03-13 | 2020-03-05 | 2020-03-06 | 4 | 5 | 5 | 4 |
| 2020-03-09 | 2020-03-06 | 2020-02-27 | 2020-02-28 | *null* | *null* | *null* | -3 |

| 2020-03-09 | 2020-03-16 | 2020-03-15 | 2020-03-09 | 4 | 6 | 6 | 7 |
|---|---|---|---|---|---|---|---|
| 2020-03-09 | 2020-03-23 | 2020-03-12 | 2020-03-16 | 8 | 10 | 11 | 14 |
| 2020-03-09 | 2020-04-10 | 2020-04-02 | 2020-04-03 | 20 | 24 | 25 | 32 |
| 2020-03-09 | 2021-03-10 | 2021-03-02 | 2021-03-03 | 210 | 262 | 263 | 366 |

# EXAMPLE - DATEDIF Function

This example illustrates how to calculate the number of days that have elapsed between the order date and today.

**Function**:

| Item | Description |
|------|-------------|
| DATEDIF Function | Calculates the difference between two valid date values for the specified units of measure. |
| TODAY Function | Derives the value for the current date in UTC time zone. You can specify a different time zone by optional parameter. |
| IF Function | The IF function allows you to build if/then/else conditional logic within your transforms. |

**Source:**

For the orders in the following set, you want to charge interest for those ones that are older than 90 days.

| OrderId | OrderDate | Amount |
|---------|-----------|--------|
| 1001 | 1/31/16 | 1000 |
| 1002 | 11/15/15 | 1000 |
| 1003 | 12/18/15 | 1000 |
| 1004 | 1/15/16 | 1000 |

**Transformation:**

The first step is to create a column containing today's (03/03/16) date value:

| Transformation Name | New formula |
|---------------------|-------------|
| **Parameter: Formula type** | Single row formula |
| **Parameter: Formula** | TODAY() |
| **Parameter: New column name** | 'Today' |

You can now use this value as the basis for computing the number of elapsed days for each invoice:

| Transformation Name | New formula |
|---------------------|-------------|
| **Parameter: Formula type** | Single row formula |
| **Parameter: Formula** | DATEDIF(OrderDate, Today, day) |

The age of each invoice in days is displayed in the new column. Now, you want to add a little bit of information to this comparison. Instead of just calculating the number of days, you could write out the action to undertake. Replace the above with the following:

| Transformation Name | New formula |
|---------------------|-------------|
| **Parameter: Formula type** | Single row formula |
| **Parameter: Formula** | IF((DATEDIF(OrderDate, Today, day) > 90),'Charge interest','no action') |

| | |
|---|---|
| **Parameter: New column name** | `'TakeAction'` |

To be fair to your customers, you might want to issue a notice at 45 days that the invoice is outstanding. You can replace the above with the following:

| | |
|---|---|
| **Transformation Name** | `New formula` |
| **Parameter: Formula type** | `Single row formula` |
| **Parameter: Formula** | `IF(DATEDIF(OrderDate, Today, day) > 90,'Charge interest',IF (DATEDIF(OrderDate, Today, day) > 45),'Send letter','no action'))` |
| **Parameter: New column name** | `'TakeAction'` |

By using nested instances of the `IF` function, you can generate multiple results in the `TakeAction` column.

For the items that are over 90 days old, you want to charge 5% interest. You can do the following:

| | |
|---|---|
| **Transformation Name** | `Edit column with formula` |
| **Parameter: Columns** | `Amount` |
| **Parameter: Formula** | `IF(TakeAction == 'Charge interest',Amount * 1.05, Amount)` |

The above sets the value in the `Amount` column based on the conditional of whether the `TakeAction` column value is `Charge interest`. If so, apply 5% interest to the value in the `Amount` column.

**Results:**

| OrderId | OrderDate | Amount | Today | TakeAction |
|---------|-----------|--------|----------|-----------------|
| 1001 | 1/31/16 | 1000 | 03/03/16 | no action |
| 1002 | 11/15/15 | 1050 | 03/03/16 | Charge interest |
| 1003 | 12/18/15 | 1000 | 03/03/16 | Send letter |
| 1004 | 1/15/16 | 1000 | 03/03/16 | Send letter |

# EXAMPLE - Date Functions

This example illustrates to how to use date-related functions to derive specific values for a Datetime column type.

**Functions**:

| Item | Description |
|------|-------------|
| YEAR Function | Derives the four-digit year value from a Datetime value. Source value can be a a reference to a column containing Datetime values or a literal. |
| MONTH Function | Derives the month integer value from a Datetime value. Source value can be a a reference to a column containing Datetime values or a literal. |
| MONTHNAME Function | Derives the full name from a Datetime value of the corresponding month as a String. Source value can be a reference to a column containing Datetime values or a literal. |
| WEEKDAYNAME Function | Derives the full name from a Datetime value of the corresponding weekday as a String. Source value can be a reference to a column containing Datetime values or a literal. |
| DAY Function | Derives the numeric day value from a Datetime value. Source value can be a a reference to a column containing Datetime values or a literal. |
| HOUR Function | Derives the hour value from a Datetime value. Generated hours are expressed according to the 24-hour clock. |
| MINUTE Function | Derives the minutes value from a Datetime value. Minutes are expressed as integers from 0 to 59. |
| SECOND Function | Derives the seconds value from a Datetime value. Source value can be a a reference to a column containing Datetime values or a literal. |

**Source:**

| date |
|------|
| 2/8/16 15:41 |
| 12/30/15 0:00 |
| 4/26/15 7:07 |

**Transformation:**

| Transformation Name | New formula |
|---------------------|-------------|
| **Parameter: Formula type** | Single row formula |
| **Parameter: Formula** | YEAR (date) |

| Transformation Name | New formula |
|---------------------|-------------|
| **Parameter: Formula type** | Single row formula |
| **Parameter: Formula** | MONTH (date) |

| Transformation Name | New formula |
|---------------------|-------------|
| **Parameter: Formula type** | Single row formula |
| **Parameter: Formula** | MONTHNAME (date) |

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | WEEKDAYNAME (date) |

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | DAY (date) |

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | HOUR (date) |

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | MINUTE (date) |

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | SECOND (date) |

**Results:**

> **NOTE:** If the source Datetime value does not contain a valid input for one of these functions, no value is returned. See the `second_date` column below.

| date | year_date | month_date | monthname_date | weekdayname_date | day_date | hour_date | minute_date | secon |
|---|---|---|---|---|---|---|---|---|
| 2/8/16 15:41 | 2016 | 2 | February | Monday | 8 | 15 | 41 | |
| 12/30 /15 0: 00 | 2015 | 12 | December | Wednesday | 30 | 0 | 0 | |
| 4/26 /15 7: 07 | 2015 | 4 | April | Sunday | 26 | 7 | 7 | |

# EXAMPLE - Date Functions - Min Max and Mode

This example shows how you can apply statistical functions on Datetime columns.

**Functions**:

| Item | Description |
|---|---|
| MINDATE Function | Computes the minimum value found in all row values in a Datetime column. |
| MAXDATE Function | Computes the maximum value found in all row values in a Datetime column. |
| MODEDATE Function | Computes the most frequent (mode) value found in all row values in a Datetime column. |

**Source:**

The following dataset contains a set of three available dates for a set of classes:

| classId | Date1 | Date2 | Date3 |
|---|---|---|---|
| c001 | 2020-03-09 | 2020-03-13 | 2020-03-17 |
| c002 | 2020-03-09 | 2020-03-06 | 2020-03-21 |
| c003 | 2020-03-09 | 2020-03-16 | 2020-03-23 |
| c004 | 2020-03-09 | 2020-03-23 | 2020-04-06 |
| c005 | 2020-03-09 | 2020-04-09 | 2020-05-09 |
| c006 | 2020-03-09 | 2020-08-09 | 2021-01-09 |

**Transformation:**

To compare dates across multiple columns, you must consolidate the values into a single column. You can use the following transformation to do so:

| | |
|---|---|
| **Transformation Name** | `Unpivot columns` |
| **Parameter: Columns** | `Date1,Date2,Date3` |
| **Parameter: Group size** | `1` |

The dataset is now contained in three columns, with descriptions listed below:

| classId | key | value |
|---|---|---|
| Same as previous. | DateX column identifier | Corresponding value from the DateX column. |

You can use the following to rename the `value` column to `eventDates`:

| | |
|---|---|
| **Transformation Name** | `Rename columns` |
| **Parameter: Option** | `Manual rename` |
| **Parameter: Column** | `value` |
| **Parameter: New column name** | `eventDates` |

Using the following transformations, you can create new columns containing the min, max, and mode values for the Datetime values in `eventDates`:

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | MINDATE(eventDates) |
| Parameter: New column name | earliestDate |

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | MAXDATE(eventDates) |
| Parameter: New column name | latestDate |

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | MODEDATE(eventDates) |
| Parameter: New column name | mostFrequentDate |

**Results:**

| classId | key | eventDates | mostFrequentDate | latestDate | earliestDate |
|---|---|---|---|---|---|
| c001 | Date1 | 2020-03-09 | 2020-03-09 | 2021-01-09 | 2020-03-06 |
| c001 | Date2 | 2020-03-13 | 2020-03-09 | 2021-01-09 | 2020-03-06 |
| c001 | Date3 | 2020-03-17 | 2020-03-09 | 2021-01-09 | 2020-03-06 |
| c002 | Date1 | 2020-03-09 | 2020-03-09 | 2021-01-09 | 2020-03-06 |
| c002 | Date2 | 2020-03-06 | 2020-03-09 | 2021-01-09 | 2020-03-06 |
| c002 | Date3 | 2020-03-21 | 2020-03-09 | 2021-01-09 | 2020-03-06 |
| c003 | Date1 | 2020-03-09 | 2020-03-09 | 2021-01-09 | 2020-03-06 |
| c003 | Date2 | 2020-03-16 | 2020-03-09 | 2021-01-09 | 2020-03-06 |
| c003 | Date3 | 2020-03-23 | 2020-03-09 | 2021-01-09 | 2020-03-06 |
| c004 | Date1 | 2020-03-09 | 2020-03-09 | 2021-01-09 | 2020-03-06 |
| c004 | Date2 | 2020-03-23 | 2020-03-09 | 2021-01-09 | 2020-03-06 |
| c004 | Date3 | 2020-04-06 | 2020-03-09 | 2021-01-09 | 2020-03-06 |
| c005 | Date1 | 2020-03-09 | 2020-03-09 | 2021-01-09 | 2020-03-06 |
| c005 | Date2 | 2020-04-09 | 2020-03-09 | 2021-01-09 | 2020-03-06 |
| c005 | Date3 | 2020-05-09 | 2020-03-09 | 2021-01-09 | 2020-03-06 |
| c006 | Date1 | 2020-03-09 | 2020-03-09 | 2021-01-09 | 2020-03-06 |
| c006 | Date2 | 2020-08-09 | 2020-03-09 | 2021-01-09 | 2020-03-06 |

| c006 | Date3 | 2021-01-09 | 2020-03-09 | 2021-01-09 | 2020-03-06 |
|------|-------|------------|------------|------------|------------|

# EXAMPLE - DATEIF Functions

This example illustrates how you can apply conditionals to calculate minimum, maximum, and most common date values.

**Functions**:

| Item | Description |
|---|---|
| MINDATEIF Function | Returns the minimum Datetime value of rows in each group that meet a specific condition. Set of values must valid Datetime values. |
| MAXDATEIF Function | Returns the maximum Datetime value of rows in each group that meet a specific condition. Set of values must valid Datetime values. |
| MODEDATEIF Function | Returns the most common Datetime value of rows in each group that meet a specific condition. Set of values must valid Datetime values. |

**Source:**

Here is some example transaction data:

| Date | Product | Units | UnitCost | OrderValue |
|---|---|---|---|---|
| 3/28/2020 | ProductA | 4 | 10.00 | 40.00 |
| 3/8/2020 | ProductB | 4 | 20.00 | 80.00 |
| 3/12/2020 | ProductC | 2 | 30.00 | 60.00 |
| 3/23/2020 | ProductA | 1 | 10.00 | 10.00 |
| 3/20/2020 | ProductB | 2 | 20.00 | 40.00 |
| 3/12/2020 | ProductC | 9 | 30.00 | 270.00 |
| 3/28/2020 | ProductA | 5 | 10.00 | 50.00 |
| 3/23/2020 | ProductB | 8 | 20.00 | 160.00 |
| 3/16/2020 | ProductC | 9 | 30.00 | 270.00 |
| 3/8/2020 | ProductA | 5 | 10.00 | 50.00 |
| 3/10/2020 | ProductB | 3 | 20.00 | 60.00 |
| 3/13/2020 | ProductC | 1 | 30.00 | 30.00 |
| 3/12/2020 | ProductA | 7 | 10.00 | 70.00 |
| 3/10/2020 | ProductB | 7 | 20.00 | 140.00 |
| 3/24/2020 | ProductC | 9 | 30.00 | 270.00 |
| 3/15/2020 | ProductA | 8 | 10.00 | 80.00 |
| 3/10/2020 | ProductB | 5 | 20.00 | 100.00 |
| 3/10/2020 | ProductC | 4 | 30.00 | 120.00 |

**Transformation and Results:**

These functions are useful for asking questions about your data. In the following, you can review specific questions and see the results immediately.

**Question 1:** What is the earliest date when a $100.00 transaction occurred?

| Transformation Name | New formula |
|---|---|

| Parameter: Formula type | Single row formula |
|---|---|
| Parameter: Formula | mindateif(Date, OrderValue > 100) |
| Parameter: New column name | 'Answers' |

**Results:** Value in Answers column: 3/10/2020

**Question 2:** What is the latest date when a $200.00 transaction occurred?

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | maxdateif(Date, OrderValue > 200) |
| Parameter: New column name | 'Answer' |

**Results:** Value in Answers column: 3/24/2020

**Question 3:** On what date did the most transactions occur this month?

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | modedateif(Date, OrderValue > 0) |
| Parameter: New column name | 'Answer' |

**Results:** Value in Answers column: 3/10/2020

# EXAMPLE - Day of Functions

This example illustrates how you can apply functions to derive day-of-week values out of a column of Datetime type.

**Functions**:

| Item | Description |
|------|-------------|
| WEEKDAY Function | Derives the numeric value for the day of the week (1, 2, etc.). Input must be a reference to a column containing Datetime values. |
| WEEKNUM Function | Derives the numeric value for the week within the year (1, 2, etc.). Input must be the output of the DATE function or a reference to a column containing Datetime values. The output of this function increments on Sunday. |
| DATEFORMAT Function | Formats a specified Datetime set of values according to the specified date format. Source values can be a reference to a column containing Datetime values. |

**Source:**

| myDate |
|--------|
| 10/30/17 |
| 10/31/17 |
| 11/1/17 |
| 11/2/17 |
| 11/3/17 |
| 11/4/17 |
| 11/5/17 |
| 11/6/17 |

**Transformation:**

The following transformation step generates a numeric value for the day of week in a new column:

| Transformation Name | New formula |
|---------------------|-------------|
| **Parameter: Formula type** | Single row formula |
| **Parameter: Formula** | WEEKDAY (myDate) |
| **Parameter: New column name** | 'weekDayNum' |

The following step generates a full text version of the name of the day of the week:

| Transformation Name | New formula |
|---------------------|-------------|
| **Parameter: Formula type** | Single row formula |
| **Parameter: Formula** | DATEFORMAT(myDate, 'EEEE') |
| **Parameter: New column name** | 'weekDayNameFull' |

The following step generates a three-letter abbreviation for the name of the day of the week:

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | DATEFORMAT(myDate, 'EEE') |
| Parameter: New column name | 'weekDayNameShort' |

The following step generates the numeric value of the week within the year:

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | WEEKNUM (myDate) |
| Parameter: New column name | 'weekNum' |

**Results:**

| myDate | weekDayNum | weekDayNameFull | weekDayNameShort | weekNum |
|---|---|---|---|---|
| 10/30/17 | 1 | Monday | Mon | 44 |
| 10/31/17 | 2 | Tuesday | Tue | 44 |
| 11/1/17 | 3 | Wednesday | Wed | 44 |
| 11/2/17 | 4 | Thursday | Thu | 44 |
| 11/3/17 | 5 | Friday | Fri | 44 |
| 11/4/17 | 6 | Saturday | Sat | 44 |
| 11/5/17 | 7 | Sunday | Sun | 45 |
| 11/6/17 | 1 | Monday | Mon | 45 |

# EXAMPLE - DEGREES and RADIANS Functions

This example illustrates to convert values from one unit of measure to the other.

**Functions:**

| Item | Description |
|------|-------------|
| DEGREES Function | Computes the degrees of an input value measuring the radians of an angle. The value can be a Decimal or Integer literal or a reference to a column containing numeric values. |
| RADIANS Function | Computes the radians of an input value measuring degrees of an angle. The value can be a Decimal or Integer literal or a reference to a column containing numeric values. |
| ROUND Function | Rounds input value to the nearest integer. Input can be an Integer, a Decimal, a column reference, or an expression. Optional second argument can be used to specify the number of digits to which to round. |

**Source:**

In this example, the source data contains information about a set of isosceles triangles. Each triangle is listed in a separate row, with the listed value as the size of the non-congruent angle in the triangle in degrees.

You must calculate the measurement of all three angles of each isosceles triangle in radians.

| triangle | a01 |
|----------|-----|
| t01 | 30 |
| t02 | 60 |
| t03 | 90 |
| t04 | 120 |
| t05 | 150 |

**Transformation:**

You can convert the value for the non-congruent angle to radians using the following:

| | |
|------|------|
| **Transformation Name** | New formula |
| **Parameter: Formula type** | Single row formula |
| **Parameter: Formula** | ROUND(RADIANS(a01), 4) |
| **Parameter: New column name** | 'r01' |

Now, calculate the value in degrees of the remaining two angles, which are congruent. Since the sum of all angles in a triangle is 180, the following formula can be applied to compute the size in degrees of each of these angles:

| | |
|------|------|
| **Transformation Name** | New formula |
| **Parameter: Formula type** | Single row formula |
| **Parameter: Formula** | (180 - a01) / 2 |
| **Parameter: New column name** | 'a02' |

Convert the above to radians:

| Transformation Name | New formula |
|---|---|
| **Parameter: Formula type** | Single row formula |
| **Parameter: Formula** | ROUND(RADIANS(a02), 4) |
| **Parameter: New column name** | 'r02' |

Create a second column for the other congruent angle:

| Transformation Name | New formula |
|---|---|
| **Parameter: Formula type** | Single row formula |
| **Parameter: Formula** | ROUND(RADIANS(a02), 4) |
| **Parameter: New column name** | 'r03' |

To check accuracy, you sum all three columns and convert to degrees:

| Transformation Name | New formula |
|---|---|
| **Parameter: Formula type** | Single row formula |
| **Parameter: Formula** | ROUND(RADIANS(a02), 4) |
| **Parameter: New column name** | 'checksum' |

**Results:**

After you delete the intermediate columns, you see the following results and determine the error in the checksum is acceptable:

| triangle | a01 | r03 | r02 | r01 | checksum |
|---|---|---|---|---|---|
| t01 | 30 | 1.3095 | 1.3095 | 0.5238 | 179.9967 |
| t02 | 60 | 1.0476 | 1.0476 | 1.0476 | 179.9967 |
| t03 | 90 | 0.7857 | 0.7857 | 1.5714 | 179.9967 |
| t04 | 120 | 0.5238 | 0.5238 | 2.0952 | 179.9967 |
| t05 | 150 | 0.2619 | 0.2619 | 2.6190 | 179.9967 |

# EXAMPLE - Delete and Keep Transforms

This examples illustrates how you can keep and delete rows from your dataset.

**Source:**

Your dataset includes the following order information. You want to edit your dataset so that:

- All orders for products that are no longer available are removed. These include the following product IDs: `P 100`, `P101`, `P102`, `P103`.
- All orders that were placed within the last 90 days are retained.

| OrderId | OrderDate | ProdId | ProductName | ProductColor | Qty | OrderValue |
|---------|-----------|--------|-------------|--------------|-----|------------|
| 1001 | 6/14/2015 | P100 | Hat | Brown | 1 | 90 |
| 1002 | 1/15/2016 | P101 | Hat | Black | 2 | 180 |
| 1003 | 11/11/2015 | P103 | Sweater | Black | 3 | 255 |
| 1004 | 8/6/2015 | P105 | Cardigan | Red | 4 | 320 |
| 1005 | 7/29/2015 | P103 | Sweeter | Black | 5 | 375 |
| 1006 | 12/1/2015 | P102 | Pants | White | 6 | 420 |
| 1007 | 12/28/2015 | P107 | T-shirt | White | 7 | 390 |
| 1008 | 1/15/2016 | P105 | Cardigan | Red | 8 | 420 |
| 1009 | 1/31/2016 | P108 | Coat | Navy | 9 | 495 |

**Transformation:**

First, you remove the orders for old products. Since the set of products is relatively small, you can start first by adding the following:

> **NOTE:** Just preview this transformation. Do not add it to your recipe yet.

| Transformation Name | Filter rows |
|---------------------|-------------|
| **Parameter: Condition** | Custom formula |
| **Parameter: Type of formula** | Custom single |
| **Parameter: Condition** | (ProdId == 'P100') |
| **Parameter: Action** | Delete matching rows |

When this step is previewed, you should notice that the top row in the above table is highlighted for removal. Notice how the transformation relies on the `ProdId` value. If you look at the `ProductName` value, you might notice that there is a misspelling in one of the affected rows, so that column is not a good one for comparison purposes.

You can add the other product IDs to the transformation in the following expansion of the transformation, in which any row that has a matching `ProdId` value is removed:

| Transformation | Filter rows |
|----------------|-------------|

| | |
|---|---|
| **Name** | |
| **Parameter: Condition** | `Custom formula` |
| **Parameter: Type of formula** | `Custom single` |
| **Parameter: Condition** | `(ProdId == 'P100' || ProdId == 'P101' || ProdId == 'P102' || ProdId == 'P103')` |
| **Parameter: Action** | `Delete matching rows` |

When the above step is added to your recipe, you should see data that looks like the following:

| OrderId | OrderDate | ProdId | ProductName | ProductColor | Qty | OrderValue |
|---|---|---|---|---|---|---|
| 1004 | 8/6/2015 | P105 | Cardigan | Red | 4 | 320 |
| 1007 | 12/28/2015 | P107 | T-shirt | White | 7 | 390 |
| 1008 | 1/15/2016 | P105 | Cardigan | Red | 8 | 420 |
| 1009 | 1/31/2016 | P108 | Coat | Navy | 9 | 495 |

Now, you can filter out of the dataset orders that are older than 90 days. First, add a column with today's date:

| Transformation Name | `New formula` |
|---|---|
| **Parameter: Formula type** | `Single row formula` |
| **Parameter: Formula** | `'2/25/16'` |
| **Parameter: New column name** | `'today'` |

Keep the rows that are within 90 days of this date using the following:

| Transformation Name | `Filter rows` |
|---|---|
| **Parameter: Condition** | `Custom formula` |
| **Parameter: Type of formula** | `Custom single` |
| **Parameter: Condition** | `datedif(OrderDate,today,day) <= 90` |
| **Parameter: Action** | `Keep matching rows` |

Don't forget to delete the `today` column, which is no longer needed:

| Transformation Name | `Delete columns` |
|---|---|
| **Parameter: Columns** | `today` |
| **Parameter: Action** | `Delete selected columns` |

**Results:**

| OrderId | OrderDate | ProdId | ProductName | ProductColor | Qty | OrderValue |
|---|---|---|---|---|---|---|
| 1007 | 12/28/2015 | P107 | T-shirt | White | 7 | 390 |
| 1008 | 1/15/2016 | P105 | Cardigan | Red | 8 | 420 |

| 1009 | 1/31/2016 | P108 | Coat | Navy | 9 | 495 |
|------|-----------|------|------|------|---|-----|

# EXAMPLE - Domain Functions

This examples illustrates how you can extract component parts of a URL using specialized functions for the URL data type.

**Functions**:

| Item | Description |
|---|---|
| DOMAIN Function | Finds the value for the domain from a valid URL. Input values must be of URL or String type. |
| SUBDOMAIN Function | Finds the value a subdomain value from a valid URL. Input values must be of URL or String type. |
| HOST Function | Finds the host value from a valid URL. Input values must be of URL or String type and can be literals or column references. |
| SUFFIX Function | Finds the suffix value after the domain from a valid URL. Input values must be of URL or String type. |
| URLPARAMS Function | Extracts the query parameters of a URL into an Object. The Object keys are the parameter's names, and its values are the parameter's values. Input values must be of URL or String type. |
| FILTEROBJEC T Function | Filters the keys and values from an Object data type column based on a specified key value. |

**Source:**

Your dataset includes the following values for URLs:

| URL |
|---|
| www.example.com |
| example.com/support |
| http://www.example.com/products/ |
| http://1.2.3.4 |
| https://www.example.com/free-download |
| https://www.example.com/about-us/careers |
| www.app.example.com |
| www.some.app.example.com |
| some.app.example.com |
| some.example.com |
| example.com |
| http://www.example.com?q1=broken%20record |
| http://www.example.com?query=khakis&app=pants |
| http://www.example.com?q1=broken%20record&q2=broken%20tape&q3=broken%20wrist |

**Transformation:**

When the above data is imported into the application, the column is recognized as a URL. All values are registered as valid, even the numeric address.

To extract the domain and subdomain values:

| Transformation Name | New formula |
|---|---|
| **Parameter: Formula type** | Single row formula |
| **Parameter: Formula** | DOMAIN(URL) |
| **Parameter: New column name** | 'domain_URL' |

| Transformation Name | New formula |
|---|---|
| **Parameter: Formula type** | Single row formula |
| **Parameter: Formula** | SUBDOMAIN(URL) |
| **Parameter: New column name** | 'subdomain_URL' |

| Transformation Name | New formula |
|---|---|
| **Parameter: Formula type** | Single row formula |
| **Parameter: Formula** | HOST(URL) |
| **Parameter: New column name** | 'host_URL' |

| Transformation Name | New formula |
|---|---|
| **Parameter: Formula type** | Single row formula |
| **Parameter: Formula** | SUFFIX(URL) |
| **Parameter: New column name** | 'suffix_URL' |

You can use the Pattern in the following transformation to extract protocol identifiers, if present, into a new column:

| Transformation Name | Extract text or pattern |
|---|---|
| **Parameter: Column to extract from** | URL |
| **Parameter: Option** | Custom text or pattern |
| **Parameter: Text to extract** | `` `{start}%*://` `` |

To clean this up, you might want to rename the column to `protocol_URL`.

To extract the path values, you can use the following regular expression:

> **NOTE:** Regular expressions are considered a developer-level method for pattern matching. Please use them with caution. See *Text Matching*.

| Transformation Name | Extract text or pattern |
|---|---|
| **Parameter: Column to extract from** | URL |

| Parameter: Option | Custom text or pattern |
|---|---|
| Parameter: Text to extract | `/[^*:\/\/]\/.*$/` |

The above transformation grabs a little too much of the URL. If you rename the column to `path_URL`, you can use the following regular expression to clean it up:

| Transformation Name | Extract text or pattern |
|---|---|
| Parameter: Column to extract from | URL |
| Parameter: Option | Custom text or pattern |
| Parameter: Text to extract | `/[!^\/].*$/` |

Delete the `path_URL` column and rename the `path_URL1` column to the deleted one. Then:

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | URLPARAMS(URL) |
| Parameter: New column name | 'urlParams' |

If you wanted to just see the values for the `q1` parameter, you could add the following:

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | FILTEROBJECT(urlParams,'q1') |
| Parameter: New column name | 'urlParam_q1' |

**Results:**

For display purposes, the results table has been broken down into separate sets of columns.

Column set 1:

| URL | host_URL | path_URL |
|---|---|---|
| www.example.com | www.example.com | |
| example.com/support | example.com | /support |
| http://www.example.com/products/ | www.example.com | /products/ |
| http://1.2.3.4 | 1.2.3.4 | |
| https://www.example.com/free-download | www.example.com | /free-download |
| https://www.example.com/about-us/careers | www.example.com | /about-us /careers |
| www.app.example.com | www.app.example.com | |
| www.some.app.example.com | www.some.app.example. | |

| | | |
|---|---|---|
| | com | |
| some.app.example.com | some.app.example.com | |
| some.example.com | some.example.com | |
| example.com | example.com | |
| http://www.example.com?q1=broken%20record | www.example.com | |
| http://www.example.com?query=khakis&app=pants | www.example.com | |
| http://www.example.com?q1=broken%20record&q2=broken%20tape&q3=broken%20wrist | www.example.com | |

Column set 2:

| URL | protocol_URL | subdomain_URL | domain_URL | suffix_URL |
|---|---|---|---|---|
| www.example.com | | www | example | com |
| example.com/support | | | example | com |
| http://www.example.com/products/ | http:// | www | example | com |
| http://1.2.3.4 | http:// | | | |
| https://www.example.com/free-download | https:// | www | example | com |
| https://www.example.com/about-us/careers | https:// | www | example | com |
| www.app.example.com | | www.app | example | com |
| www.some.app.example.com | | www.some.app | example | com |
| some.app.example.com | | some.app | example | com |
| some.example.com | | some | example | com |
| example.com | | | example | com |
| http://www.example.com?q1=broken%20record | http:// | www | example | com |
| http://www.example.com?query=khakis&app=pants | http:// | www | example | com |
| http://www.example.com?q1=broken%20record&q2=broken%20tape&q3=broken%20wrist | http:// | www | example | com |

Column set 3:

| URL | urlParams | urlParam_q1 |
|---|---|---|
| www.example.com | | |
| example.com/support | | |
| http://www.example.com/products/ | | |
| http://1.2.3.4 | | |
| https://www.example.com/free-download | | |
| https://www.example.com/about-us/careers | | |
| www.app.example.com | | |
| www.some.app.example.com | | |
| some.app.example.com | | |
| some.example.com | | |
| example.com | | |
| http://www.example.com?q1=broken%20record | {"q1":"broken record"} | {"q1":"broken |

| | | record"} |
|---|---|---|
| http://www.example.com?query=khakis&app=pants | {"query":"khakis","app":"pants"} | |
| http://www.example.com?q1=broken%20record&q2=broken%20tape&q3=broken%20wrist | {"q1":"broken record", "q2":"broken tape", "q3":"broken wrist"} | {"q1":"broken record"} |

# EXAMPLE - Double Metaphone Functions

This example illustrates how to use double metaphone functions to generate phonetic spellings in Designer Cloud Powered by Trifacta® Enterprise Edition.

**Functions**:

| Item | Description |
|---|---|
| DOUBLEMETAPHONE Function | Returns a two-element array of primary and secondary phonetic encodings for an input string, based on the Double Metaphone algorithm. |
| DOUBLEMETAPHONEEQUALS Function | Compares two input strings using the Double Metaphone algorithm. An optional threshold parameter can be modified to adjust the tolerance for matching. |

**Source:**

The following table contains some example strings to be compared.

| string1 | string2 | notes |
|---|---|---|
| My String | my string | comparison is case-insensitive |
| judge | juge | typo |
| knock | nock | silent letters |
| white | wite | missing letters |
| record | record | two different words in English but match the same |
| pair | pear | these match but are different words. |
| bookkeeper | book keeper | spaces cause failures in comparison |
| test1 | test123 | digits are not compared |
| the end. | the end…. | punctuation differences do not matter. |
| a elephant | an elephant | a and an are treated differently. |

**Transformation:**

You can use the `DOUBLEMETAPHONE` function to generate phonetic spellings, as in the following:

| Transformation Name | New formula |
|---|---|
| **Parameter: Formula type** | Single row formula |
| **Parameter: Formula** | DOUBLEMETAPHONE(string1) |
| **Parameter: New column name** | 'dblmeta_s1' |

You can compare `string1` and `string2` using the `DOUBLEMETAPHONEEQUALS` function:

| Transformation Name | New formula |
|---|---|
| **Parameter: Formula type** | Single row formula |
| **Parameter: Formula** | DOUBLEMETAPHONEEQUALS(string1, string2, 'normal') |
| **Parameter: New column** | 'compare' |

**Results:**

The following table contains some example strings to be compared.

| string1 | dblmeta_s1 | string2 | compare | Notes |
|---------|-----------|---------|---------|-------|
| My String | ["MSTRNK","MSTRNK"] | my string | TRUE | comparison is case-insensitive |
| judge | ["JJ","AJ"] | juge | TRUE | typo |
| knock | ["NK","NK"] | nock | TRUE | silent letters |
| white | ["AT","AT"] | wite | TRUE | missing letters |
| record | ["RKRT","RKRT"] | record | TRUE | two different words in English but match the same |
| pair | ["PR","PR"] | pear | TRUE | these match but are different words. |
| bookkeeper | ["PKPR","PKPR"] | book keeper | FALSE | spaces cause failures in comparison |
| test1 | ["TST","TST"] | test123 | TRUE | digits are not compared |
| the end. | ["0NT","TNT"] | the endâ€¦. | TRUE | punctuation differences do not matter. |
| a elephant | ["ALFNT","ALFNT"] | an elephant | FALSE | a and an are treated differently. |

# EXAMPLE - Exponentional Functions

This example demonstrates the exponential functions.

**Functions**:

| Item | Description |
|------|-------------|
| EXP Function | Computes the value of *e* raised to the specified power. The value can be a Decimal or Integer literal or a reference to a column containing numeric values. |
| LN Function | Computes the natural logarithm of an input value. The value can be a Decimal or Integer literal or a reference to a column containing numeric values. |
| LOG Function | Computes the logarithm of the first argument with a base of the second argument. |
| POW Function | Computes the value of the first argument raised to the value of the second argument. |

**Source:**

| rowNum | X |
|--------|-----|
| 1 | -2 |
| 2 | 1 |
| 3 | 0 |
| 4 | 1 |
| 5 | 2 |
| 6 | 3 |
| 7 | 4 |
| 8 | 5 |

**Transformation:**

| | |
|------|------|
| **Transformation Name** | New formula |
| **Parameter: Formula type** | Single row formula |
| **Parameter: Formula** | EXP (X) |
| **Parameter: New column name** | 'expX' |

| | |
|------|------|
| **Transformation Name** | New formula |
| **Parameter: Formula type** | Single row formula |
| **Parameter: Formula** | LN (expX) |
| **Parameter: New column name** | 'ln_expX' |

| | |
|------|------|
| **Transformation Name** | New formula |

| Parameter: Formula type | Single row formula |
|---|---|
| Parameter: Formula | LOG (X) |
| Parameter: New column name | 'logX' |

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | POW (10,logX) |
| Parameter: New column name | 'pow_logX' |

**Results:**

In the following, (null value) indicates that a null value is generated for the computation.

| rowNum | X | expX | ln_expX | logX | pow_logX |
|---|---|---|---|---|---|
| 1 | -2 | 0.1353352832366127 | -2 | (null value) | (null value) |
| 2 | -1 | 0.1353352832366127 | -0.9999999999999998 | (null value) | (null value) |
| 3 | 0 | 1 | 0 | (null value) | 0 |
| 4 | 1 | 2.718281828459045 | 1 | 0 | 1 |
| 5 | 2 | 7.3890560989306495 | 2 | 0.30102999566398114 | 1.9999999999999998 |
| 6 | 3 | 20.085536923187668 | 3 | 0.47712125471966244 | 3 |
| 7 | 4 | 54.59815003314423 | 4 | 0.6020599913279623 | 3.999999999999999 |
| 8 | 5 | 148.41315910257657 | 5 | 0.6989700043360187 | 4.999999999999999 |

# EXAMPLE - Extractkv and Unnest Transforms

This example shows how you can unpack data nested in an Object into separate columns.

**Source:**

You have the following information on used cars. The `VIN` column contains vehicle identifiers, and the `Properties` column contains key-value pairs describing characteristics of each vehicle. You want to unpack this data into separate columns.

| VIN | Properties |
|---|---|
| XX3 JT4522 | year=2004,make=Subaru,model=Impreza,color=green,mileage=125422,cost=3199 |
| HT4 UJ9122 | year=2006,make=VW,model=Passat,color=silver,mileage=102941,cost=4599 |
| KC2 WZ9231 | year=2009,make=GMC,model=Yukon,color=black,mileage=68213,cost=12899 |
| LL8 UH4921 | year=2011,make=BMW,model=328i,color=brown,mileage=57212,cost=16999 |

**Transformation:**

Add the following transformation, which identifies all of the key values in the column as beginning with alphabetical characters.

- The `valueafter` string identifies where the corresponding value begins after the key.
- The `delimiter` string indicates the end of each key-value pair.

| Transformation Name | Convert keys/values into Objects |
|---|---|
| **Parameter: Column** | `Properties` |
| **Parameter: Key** | `` `{alpha}+` `` |
| **Parameter: Separator between key and value** | `` `=` `` |
| **Parameter: Delimiter between pair** | `','` |

Now that the Object of values has been created, you can use the `unnest` transform to unpack this mapped data. In the following, each key is specified, which results in separate columns headed by the named key:

> **NOTE**: Each key must be entered on a separate line in the Path to elements area.

| Transformation Name | Unnest Objects into columns |
|---|---|
| **Parameter: Column** | `extractkv_Properties` |
| **Parameter: Paths to elements** | `year` |
| **Parameter: Paths to elements** | `make` |
| **Parameter: Paths to elements** | `model` |
| **Parameter: Paths to elements** | `color` |
| **Parameter: Paths to elements** | `mileage` |
| **Parameter: Paths to elements** | `cost` |

**Results:**

When you delete the unnecessary Properties columns, the dataset now looks like the following:

| VIN | year | make | model | color | mileage | cost |
|-----|------|------|-------|-------|---------|------|
| XX3 JT4522 | 2004 | Subaru | Impreza | green | 125422 | 3199 |
| HT4 UJ9122 | 2006 | VW | Passat | silver | 102941 | 4599 |
| KC2 WZ9231 | 2009 | GMC | Yukon | black | 68213 | 12899 |
| LL8 UH4921 | 2011 | BMW | 328i | brown | 57212 | 16999 |

# EXAMPLE - Extractlist Transform

This example illustrates how to extract values from a column.

**Source:**

The following dataset contains counts of support emails processed by each member of the support team for individual customers over a six-month period. In this case, you are interested in the total number of emails processed for each customer.

Unfortunately, the data is ragged, as there are no entries for a support team member if he or she has not answered an email for a customer.

| custId | startDate | endDate | supportEmailCount |
|--------|-----------|---------|-------------------|
| C001 | 7/15/2015 | 12/31/2015 | ["Max":"2","Ted":"0","Sally":"12","Jack":"6","Sue":"4"] |
| C002 | 7/15/2015 | 12/31/2015 | ["Sally":"4","Sue":"3"] |
| C003 | 7/15/2015 | 12/31/2015 | ["Ted":"12","Sally":"2"] |
| C004 | 7/15/2015 | 12/31/2015 | ["Jack":"7","Sue":"4","Ted":"5"] |

If the data is imported from a CSV file, you might need to make some simple Replace Text or Pattern transformations to clean up the data to look like the above example.

**Transformation:**

Use the following transformation to extract just the numeric values from the `supportEmailCount` array:

| Transformation Name | Extract matches into Array |
|---------------------|----------------------------|
| **Parameter: Column** | `supportEmailCount` |
| **Parameter: Pattern matching elements in list** | `` `{digit}+` `` |

You should now have a column `extractlist_supportEmailCount` containing a ragged array. You can use the following transformations to convert this data to a comma-separated list of values:

| Transformation Name | Replace text or pattern |
|---------------------|-------------------------|
| **Parameter: Column** | `extractlist_supportEmailCount` |
| **Parameter: Find** | `` `[` `` |
| **Parameter: Replace with** | `''` |
| **Parameter: Match all occurrences** | `true` |

| Transformation Name | Replace text or pattern |
|---------------------|-------------------------|
| **Parameter: Column** | `extractlist_supportEmailCount` |
| **Parameter: Find** | `` `]` `` |

| | |
|---|---|
| **Parameter: Replace with** | `''` |
| **Parameter: Match all occurrences** | `true` |

| | |
|---|---|
| **Transformation Name** | `Replace text or pattern` |
| **Parameter: Column** | `extractlist_supportEmailCount` |
| **Parameter: Find** | `` `"` `` |
| **Parameter: Replace with** | `''` |
| **Parameter: Match all occurrences** | `true` |

Convert the column to String data type.

You can now split out the column into separate columns containing individual values in the modified source. The `l imit` parameter specifies the number of splits to create, resulting in 5 new columns, which is the maximum number of entries in the source arrays.

| | |
|---|---|
| **Transformation Name** | `Split by delimiter` |
| **Parameter: Column** | `extractlist_supportEmailCount` |
| **Parameter: Option** | `On pattern` |
| **Parameter: Match pattern** | `','` |
| **Parameter: Number of columns to create** | `4` |

You might have to set the type for each generated column to Integer. If you try to use a New Formula transformation to calculate the sum of all of the generated columns, it only returns values for the first row because the missing rows are null values.

In the columns containing null values, select the missing value bar in the data histogram. Select the Replace suggestion card, and modify the transformation to write a `0` in place of the null value, as follows:

| | |
|---|---|
| **Transformation Name** | `Edit column with formula` |
| **Parameter: Columns** | `extractlist_supportEmailCount3` |
| **Parameter: Formula** | `'0'` |
| **Parameter: Group rows by** | `ismissing([extractlist_supportEmailCount3])` |

Repeat this step for any other column containing null values.

You can now use the following to sum the values in the generated columns:

| | |
|---|---|
| **Transformation Name** | `New formula` |
| **Parameter: Formula type** | `Single row formula` |

| Parameter: Formula | `(extractlist_supportEmailCount1 + extractlist_supportEmailCount2 + extractlist_supportEmailCount3 + extractlist_supportEmailCount4 + extractlist_supportEmailCount5)` |
|---|---|

**Results:**

After renaming the generated column to `totalSupportEmails` and dropping the columns used to create it, your dataset should look like the following:

| custId | startDate | endDate | supportEmailCount | totalSupportEmails |
|---|---|---|---|---|
| C001 | 7/15/2015 | 12/31/2015 | ["Max":"2","Ted":"0","Sally":"12","Jack":"6","Sue":"4"] | 24 |
| C002 | 7/15/2015 | 12/31/2015 | ["Sally":"4","Sue":"3"] | 7 |
| C003 | 7/15/2015 | 12/31/2015 | ["Ted":"12","Sally":"2"] | 14 |
| C004 | 7/15/2015 | 12/31/2015 | ["Jack":"7","Sue":"4","Ted":"5"] | 16 |

# EXAMPLE - Extract Values

In this example, you extract one or more values from a source column and assemble them in an Array column.

Suppose you need to extract the hashtags from customer tweets to another column. In such cases, you can use the `{hashtag}` Trifacta pattern to extract all hashtag values from a customer's tweets into a new column.

**Source**:

The following dataset contains customer tweets across different locations.

| User Name | Location | Customer tweets |
|-----------|----------|-----------------|
| James | U.K | Excited to announce that we've transitioned Wrangler from a hybrid desktop application to a completely cloud-based service! #dataprep #businessintelligence #CommitToCleanData # London |
| Mark | Berlin | Learnt more about the importance of identifying issues in your data—early and often #CommitToCleanData #predictivetransformations #realbusinessintelligence |
| Catherine | Paris | Clean data is the foundation of your analysis. Learn more about what we consider the five tenets of sound #dataprep, starting with #1a prioritizing and setting targets.  #startwiththeuser #realbusinessintelligence #Paris |
| Dave | New York | Learn how #NewYorklife<br><br>onboarded as part of their #bigdata  #dataprep initiative to unlock hidden insights and make them accessible across departments. |
| Christy | San Francisco | How can you quickly determine the number of times a user ID appears in your data?#dataprep #pivot #aggregation#machinelearning initiatives #SFO |

**Transformation**:

The following transformation extracts the hashtag messages from customer tweets.

| Transformation Name | `Extract matches into Array` |
|---------------------|------------------------------|
| **Parameter: Column** | `customer_tweets` |
| **Parameter: Pattern matching elements in the list** | `` `{hashtag}` `` |
| **Parameter: New column name** | `Hashtag tweets` |

Then, the source column can be deleted.

**Results**:

| User Name | Location | Hashtag tweets |
|-----------|----------|----------------|
| James | U.K | ["#dataprep", "#businessintelligence", "#CommitToCleanData", " # London"] |
| Mark | Berlin | ["#CommitToCleanData",  "#predictivetransformations", "#realbusinessintelligence", "0"] |
| Catherine | Paris | ["#dataprep", "#startwiththeuser","#realbusinessintelligence", "# Paris"] |
| Dave | New York | ["#NewYorklife", "dataprep", "bigdata", "0"] |
| Christy | SanFrancisco | [ "dataprep", "#pivot", "#aggregation", "#machinelearning"] |

# EXAMPLE - Flatten an Array

This section describes how to flatten the values in an Array into separate rows in your dataset.

**Source:**

In the following example dataset, students took the same test three times, and their scores were stored in any array in the `Scores` column.

| LastName | FirstName | Scores |
|----------|-----------|--------|
| Adams | Allen | [81,87,83,79] |
| Burns | Bonnie | [98,94,92,85] |
| Cannon | Chris | [88,81,85,78] |

**Transformation:**

When the data is imported, you might have to re-type the `Scores` column as an array:

| Transformation Name | `Change column data type` |
|---------------------|---------------------------|
| **Parameter: Columns** | `Scores` |
| **Parameter: New type** | `Array` |

You can now flatten the `Scores` column data into separate rows:

| Transformation Name | `Expand Array into rows` |
|---------------------|---------------------------|
| **Parameter: Column** | `Scores` |

**Results:**

| LastName | FirstName | Scores |
|----------|-----------|--------|
| Adams | Allen | 81 |
| Adams | Allen | 87 |
| Adams | Allen | 83 |
| Adams | Allen | 79 |
| Burns | Bonnie | 98 |
| Burns | Bonnie | 94 |
| Burns | Bonnie | 92 |
| Burns | Bonnie | 85 |
| Cannon | Chris | 88 |
| Cannon | Chris | 81 |
| Cannon | Chris | 85 |
| Cannon | Chris | 78 |

> **Tip:** You can use aggregation functions on the above data to complete values like average, minimum, and maximum scores. When these aggregation calculations are grouped by student, you can perform the calculations for each student.

# EXAMPLE - Flatten and Unnest Transforms

This example illustrates you to use the flatten and unnest transforms.

**Source:**

You have the following data on student test scores. Scores on individual scores are stored in the `Scores` array, and you need to be able to track each test on a uniquely identifiable row. This example has two goals:

1. One row for each student test
2. Unique identifier for each student-score combination

| LastName | FirstName | Scores |
|----------|-----------|--------|
| Adams | Allen | [81,87,83,79] |
| Burns | Bonnie | [98,94,92,85] |
| Cannon | Charles | [88,81,85,78] |

**Transformation:**

When the data is imported from CSV format, you must add a `header` transform and remove the quotes from the `Scores` column:

| | |
|---|---|
| **Transformation Name** | Rename column with row(s) |
| **Parameter: Option** | Use row(s) as column names |
| **Parameter: Type** | Use a single row to name columns |
| **Parameter: Row number** | 1 |

| | |
|---|---|
| **Transformation Name** | Replace text or pattern |
| **Parameter: Column** | colScores |
| **Parameter: Find** | '\"' |
| **Parameter: Replace with** | '' |
| **Parameter: Match all occurrences** | true |

**Validate test date:** To begin, you might want to check to see if you have the proper number of test scores for each student. You can use the following transform to calculate the difference between the expected number of elements in the `Scores` array (4) and the actual number:

| | |
|---|---|
| **Transformation Name** | New formula |
| **Parameter: Formula type** | Single row formula |
| **Parameter: Formula** | (4 - arraylen(Scores)) |
| **Parameter: New column name** | 'numMissingTests' |

When the transform is previewed, you can see in the sample dataset that all tests are included. You might or might not want to include this column in the final dataset, as you might identify missing tests when the recipe is run at scale.

**Unique row identifier:** The `Scores` array must be broken out into individual rows for each test. However, there is no unique identifier for the row to track individual tests. In theory, you could use the combination of `LastName-FirstName-Scores` values to do so, but if a student recorded the same score twice, your dataset has duplicate rows. In the following transform, you create a parallel array called `Tests`, which contains an index array for the number of values in the `Scores` column. Index values start at `0`:

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | range(0,arraylen(Scores)) |
| Parameter: New column name | 'Tests' |

Also, we will want to create an identifier for the source row using the `sourcerownumber` function:

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | sourcerownumber() |
| Parameter: New column name | 'orderIndex' |

**One row for each student test:** Your data should look like the following:

| LastName | FirstName | Scores | Tests | orderIndex |
|---|---|---|---|---|
| Adams | Allen | [81,87,83,79] | [0,1,2,3] | 2 |
| Burns | Bonnie | [98,94,92,85] | [0,1,2,3] | 3 |
| Cannon | Charles | [88,81,85,78] | [0,1,2,3] | 4 |

Now, you want to bring together the `Tests` and `Scores` arrays into a single nested array using the `arrayzip` function:

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | arrayzip([Tests,Scores]) |

Your dataset has been changed:

| LastName | FirstName | Scores | Tests | orderIndex | column1 |
|---|---|---|---|---|---|
| Adams | Allen | [81,87,83,79] | [0,1,2,3] | 2 | [[0,81],[1,87],[2,83],[3,79]] |
| Adams | Bonnie | [98,94,92,85] | [0,1,2,3] | 3 | [[0,98],[1,94],[2,92],[3,85]] |
| Cannon | Charles | [88,81,85,78] | [0,1,2,3] | 4 | [[0,88],[1,81],[2,85],[3,78]] |

Use the following to unpack the nested array:

| Transformation Name | Expand arrays to rows |
|---|---|
| Parameter: Column | column1 |

Each test-score combination is now broken out into a separate row. The nested Test-Score combinations must be broken out into separate columns using the following:

| Transformation Name | Unnest Objects into columns |
|---|---|
| Parameter: Column | column1 |
| Parameter: Paths to elements | '[0]','[1]' |

After you delete `column1`, which is no longer needed you should rename the two generated columns:

| Transformation Name | Rename columns |
|---|---|
| Parameter: Option | Manual rename |
| Parameter: Column | column_0 |
| Parameter: New column name | 'TestNum' |

| Transformation Name | Rename columns |
|---|---|
| Parameter: Option | Manual rename |
| Parameter: Column | column_1 |
| Parameter: New column name | 'TestScore' |

**Unique row identifier:** You can do one more step to create unique test identifiers, which identify the specific test for each student. The following uses the original row identifier `OrderIndex` as an identifier for the student and the `TestNumber` value to create the `TestId` column value:

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | (orderIndex * 10) + TestNum |
| Parameter: New column name | 'TestId' |

The above are integer values. To make your identifiers look prettier, you might add the following:

| Transformation Name | Merge columns |
|---|---|
| Parameter: Columns | 'TestId00','TestId' |

**Extending:** You might want to generate some summary statistical information on this dataset. For example, you might be interested in calculating each student's average test score. This step requires figuring out how to properly group the test values. In this case, you cannot group by the `LastName` value, and when executed at scale, there might be collisions between first names when this recipe is run at scale. So, you might need to create a kind of primary key using the following:

| Transformation Name | Merge columns |
|---|---|
| **Parameter: Columns** | `'LastName','FirstName'` |
| **Parameter: Separator** | `'-'` |
| **Parameter: New column name** | `'studentId'` |

You can now use this as a grouping parameter for your calculation:

| Transformation Name | New formula |
|---|---|
| **Parameter: Formula type** | `Single row formula` |
| **Parameter: Formula** | `average(TestScore)` |
| **Parameter: Group rows by** | `studentId` |
| **Parameter: New column name** | `'avg_TestScore'` |

**Results:**

After you delete unnecessary columns and move your columns around, the dataset should look like the following:

| TestId | LastName | FirstName | TestNum | TestScore | studentId | avg_TestScore |
|---|---|---|---|---|---|---|
| TestId0021 | Adams | Allen | 0 | 81 | Adams-Allen | 82.5 |
| TestId0022 | Adams | Allen | 1 | 87 | Adams-Allen | 82.5 |
| TestId0023 | Adams | Allen | 2 | 83 | Adams-Allen | 82.5 |
| TestId0024 | Adams | Allen | 3 | 79 | Adams-Allen | 82.5 |
| TestId0031 | Adams | Bonnie | 0 | 98 | Adams-Bonnie | 92.25 |
| TestId0032 | Adams | Bonnie | 1 | 94 | Adams-Bonnie | 92.25 |
| TestId0033 | Adams | Bonnie | 2 | 92 | Adams-Bonnie | 92.25 |
| TestId0034 | Adams | Bonnie | 3 | 85 | Adams-Bonnie | 92.25 |
| TestId0041 | Cannon | Chris | 0 | 88 | Cannon-Chris | 83 |
| TestId0042 | Cannon | Chris | 1 | 81 | Cannon-Chris | 83 |
| TestId0043 | Cannon | Chris | 2 | 85 | Cannon-Chris | 83 |
| TestId0044 | Cannon | Chris | 3 | 78 | Cannon-Chris | 83 |

# EXAMPLE - Flatten and Valuestocols Transforms

This example shows how you can break out a column of nested values into separate rows and columns of data.

**Source:**

The following data covers magazine subscriptions for individual customers. Their subscriptions are stored in an array of values. You are interested in who is subscribing to each magazine.

| CustId | Subscriptions |
|---|---|
| Anne Aimes | ["Little House and Garden","Sporty Pants","Life on the Range"] |
| Barry Barnes | ["Sporty Pants","Investing Smart"] |
| Cindy Compton | ["Cakes n Pies","Powerlifting Plus","Running Days"] |
| Darryl Diaz | ["Investing Smart","Cakes n Pies"] |

**Transformation:**

When this data is loaded into the Transformer, you might need to apply a `header` to it. If it is in CSV format, you might need to apply some `replace` transformations to clean up the `Subscriptions` column so it looks like the above.

When the `Subscriptions` column contains cleanly formatted arrays, the column is re-typed as Array type. You can then apply the following transformation:

| Transformation Name | Expand Array into rows |
|---|---|
| **Parameter: Column** | Subscriptions |

Each `CustId`/`Subscription` combination is now written to a separate row. You can use this new data structure to break out instances of magazine subscriptions. Using the following transformation, you can add the corresponding `CustId` value to the column:

| Transformation Name | Convert values to columns |
|---|---|
| **Parameter: Column** | Subscriptions |
| **Parameter: Fill when present** | CustId |

Delete the two source columns:

| Transformation Name | Delete columns |
|---|---|
| **Parameter: Columns** | CustId,Subscriptions |
| **Parameter: Action** | Delete selected columns |

**Results:**

| Little_House_and_Garden | Sporty_Pants | Life_on_the_Range | Investing_Smart | Cakes_n_Pies | Powerlifting_Plus | R |
|---|---|---|---|---|---|---|
| Anne Aimes | | | | | | |
| | Anne Aimes | | | | | |
| | | Anne Aimes | | | | |

| | | | | | | |
|---|---|---|---|---|---|---|
| | Barry Barnes | | | | | |
| | | | Barry Barnes | | | |
| | | | | Cindy Compton | | |
| | | | | | Cindy Compton | |
| | | | | | | Cir |
| | | | Darryl Diaz | | | |
| | | | | Darry Diaz | | |

# EXAMPLE - IF Data Type Validation Functions

This example illustrates how to use the IF* functions for data type validation.

**Functions**:

| Item | Description |
|------|-------------|
| IFNULL Function | The `IFNULL` function writes out a specified value if the source value is a null. Otherwise, it writes the source value. Input can be a literal, a column reference, or a function. |
| IFMISSING Function | The `IFMISSING` function writes out a specified value if the source value is a null or missing value. Otherwise, it writes the source value. Input can be a literal, a column reference, or a function. |
| IFMISMATCHED Function | The `IFMISMATCHED` function writes out a specified value if the input expression does not match the specified data type or typing array. Otherwise, it writes the source value. Input can be a literal, a column reference, or a function. |
| IFVALID Function | The `IFVALID` function writes out a specified value if the input expression matches the specified data type. Otherwise, it writes the source value. Input can be a literal, a column reference, or a function. |
| MERGE Function | Merges two or more columns of String type to generate output of String type. Optionally, you can insert a delimiter between the merged values. |

**Source:**

The following simple table lists zip codes by customer identifier:

| custId | custZip |
|--------|---------|
| C001 | 98123 |
| C002 | 94105 |
| C003 | 12415 |
| C004 | 12451-2234 |
| C005 | 12441-298 |
| C006 | |
| C007 | |
| C008 | 1242 |
| C009 | 1104 |

**Transformation:**

When the above is imported into the Transformer page, you notice the following:

- The `custZip` column is typed as Integer.
- There are two missing and two mismatched values in the `custZip` column.

First, you test for valid values in the `custZip` column. Using the `IFVALID` function, you can validate against any data type:

| Transformation Name | New formula |
|---------------------|-------------|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | IFVALID(custZip, 'Zipcode', 'ok') |
| Parameter: New column | 'status' |

| | |
|---|---|
| name | |

**Fix four-digit zips:** In the `status` column are instances of `ok` for the top four rows. You notice that the bottom two rows contain four-digit codes.

Since the `custZip` values were originally imported as Integer, any leading `0` values are deleted. In this case, you can add back the leading zero. Before the previous step, change the data type of `zip` to String and insert the following:

| | |
|---|---|
| **Transformation Name** | New formula |
| **Parameter: Formula type** | Single row formula |
| **Parameter: Formula** | IF(LEN(custZip)==4,'0','') |
| **Parameter: New column name** | 'FourDigitZip' |

| | |
|---|---|
| **Transformation Name** | New formula |
| **Parameter: Formula type** | Single row formula |
| **Parameter: Formula** | MERGE([FourDigitZip,custZip]) |
| **Parameter: New column name** | 'custZip2' |

| | |
|---|---|
| **Transformation Name** | Edit column with formula |
| **Parameter: Columns** | zip |
| **Parameter: Formula** | custZip2 |

| | |
|---|---|
| **Transformation Name** | Delete columns |
| **Parameter: Columns** | FourDigitZip,custZip2 |
| **Parameter: Action** | Delete selected columns |

Now, when you click the last recipe step, you should see that two more rows in `status` are listed as `Ok`.

For the zip code with the three-digit extension, you can simply remove that extension to make it valid. Click the step above the last one. In the data grid, highlight the value. Click the Replace suggestion card. Select the option that uses the following for the matching pattern:

```
'-{digit}{3}{end}'
```

The above means that all three-digit extensions are deleted from the zip. You can do the same for any two- and one-digit extensions, although there are none in this sample.

**Missing and null values:** Now, you need to address how to handle missing and null values. The `IFMISSING` tests for both missing and null values, while the `IFNULL` tests just for null values. In this example, you want to delete null values, which could mean that the data for that row is malformed and to write a status of `missing` for missing values.

Click above the last line in the recipe to insert the following:

| | |
|---|---|
| | |

| Transformation Name | Edit column with formula |
|---|---|
| Parameter: Columns | custZip |
| Parameter: Formula | IFNULL(custZip, 'xxxxx') |

| Transformation Name | Edit column with formula |
|---|---|
| Parameter: Columns | custZip |
| Parameter: Formula | IFMISSING(custZip, '00000') |

Now, when you click the last line of the recipe, only the null value is listed as having a status other than `ok`. You can use the following to remove this row and all like it:

| Transformation Name | Filter rows |
|---|---|
| Parameter: Condition | Custom formula |
| Parameter: Type of formula | Custom single |
| Parameter: Condition | (status == 'xxxxx') |
| Parameter: Action | Delete matching rows |

### Results:

| custId | custZip | status |
|---|---|---|
| C001 | 98123 | ok |
| C002 | 94105 | ok |
| C003 | 12415 | ok |
| C004 | 12451-2234 | ok |
| C005 | 12441-298 | ok |
| C006 | 00000 | ok |
| C008 | 1242 | ok |
| C009 | 1104 | ok |

As an exercise, you might repeat the above steps starting with the `IFMISMATCHED` function determining the value in the `status` column:

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | IFMISMATCHED(custZip, 'Zipcode', 'mismatched') |
| Parameter: New column name | 'status' |

# EXAMPLE - IPTOINT Function

This examples illustrates how you can convert IP addresses to numeric values for purposes of comparison and sorting.

**Functions**:

| Item | Description |
|------|-------------|
| IPTOINT Function | Computes an integer value for a four-octet internet protocol (IP) address. Source value must be a valid IP address or a column reference to IP addresses. |
| IPFROMINT Function | Computes a four-octet internet protocol (IP) address from a 32-bit integer input. |

**Source:**

Your dataset includes the following values for IP addresses:

| IpAddr |
|--------|
| 192.0.0.1 |
| 10.10.10.10 |
| 1.2.3.4 |
| 1.2.3 |
| http://12.13.14.15 |
| https://16.17.18.19 |

**Transformation:**

When the above data is imported, the application initially types the column as URL values, due to the presence of the `http://` and `https://` protocol identifiers. Select the IP Address data type for the column. The last three values are listed as mismatched values. You can fix the issues with the last two entries by applying the following transform, which matches on both `http://` and `https://` strings:

| Transformation Name | Replace text or pattern |
|---------------------|-------------------------|
| Parameter: Column | IpAddr |
| Parameter: Find | `` `http%?://` `` |
| Parameter: Replace with | `''` |

> **NOTE:** The `%?` Pattern matches zero or one time on any character, which enables the matching on both variants of the protocol identifier.

Now, only the `1.2.3` value is mismatched. Perhaps you know that there is a missing zero at the end of it. To add it back, you can do the following:

| Transformation Name | Replace text or pattern |
|---------------------|-------------------------|
| Parameter: Column | IpAddr |
| Parameter: Find | `` `1.2.3[end]` `` |
| Parameter: Replace with | `'1.2.3.0'` |

| | |
|---|---|
| **Parameter: Match all occurrences** | `true` |

All values in the column should be valid for the IP Address data type. To convert these values to their integer equivalents:

| | |
|---|---|
| **Transformation Name** | `New formula` |
| **Parameter: Formula type** | `Single row formula` |
| **Parameter: Formula** | `IPTOINT(IpAddr)` |
| **Parameter: New column name** | `'ip_as_int'` |

You can now manipulate the data based on this numeric key. To convert the integer values back to IP addresses for checking purposes, use the following:

| | |
|---|---|
| **Transformation Name** | `New formula` |
| **Parameter: Formula type** | `Single row formula` |
| **Parameter: Formula** | `IPFROMINT(ip_as_int)` |
| **Parameter: New column name** | `'ip_check'` |

**Results:**

| X | ip_as_int | ip_check |
|---|---|---|
| 192.0.0.1 | 3221225473 | 192.0.0.1 |
| 10.10.10.10 | 168430090 | 10.10.10.10 |
| 1.2.3.4 | 16909060 | 1.2.3.4 |
| 1.2.3.0 | 16909056 | 1.2.3.0 |
| 12.13.14.15 | 202182159 | 12.13.14.15 |
| 16.17.18.19 | 269554195 | 16.17.18.19 |

# EXAMPLE - KTHLARGESTDATE Functions

This example illustrates how you can apply conditionals to calculate minimum, maximum, and most common date values.

**Functions**:

| Item | Description |
|---|---|
| KTHLARGESTDATE Function | Extracts the ranked Datetime value from the values in a column, where $k=1$ returns the maximum value. The value for $k$ must be between 1 and 1000, inclusive. Inputs must be valid Datetime values. |
| KTHLARGESTUNIQUEDATE Function | Extracts the ranked unique Datetime value from the values in a column, where $k=1$ returns the maximum value. The value for $k$ must be between 1 and 1000, inclusive. Inputs must be Datetime. |
| KTHLARGESTDATEIF Function | Extracts the ranked Datetime value from the values in a column, where $k=1$ returns the maximum value, when a specified condition is met. The value for $k$ must be between 1 and 1000, inclusive. Inputs must be Datetime. |
| KTHLARGESTUNIQUEDATEIF Function | Extracts the ranked unique Datetime value from the values in a column, where $k=1$ returns the maximum value, when a specified condition is met. The value for $k$ must be between 1 and 1000, inclusive. Inputs must be Datetime. |

**Source:**

Here is some example transaction data:

| Date | Product | Units | UnitCost | OrderValue |
|---|---|---|---|---|
| 3/28/2020 | ProductA | 4 | 10.00 | 40.00 |
| 3/8/2020 | ProductB | 4 | 20.00 | 80.00 |
| 3/12/2020 | ProductC | 2 | 30.00 | 60.00 |
| 3/23/2020 | ProductA | 1 | 10.00 | 10.00 |
| 3/20/2020 | ProductB | 2 | 20.00 | 40.00 |
| 3/12/2020 | ProductC | 9 | 30.00 | 270.00 |
| 3/28/2020 | ProductA | 5 | 10.00 | 50.00 |
| 3/23/2020 | ProductB | 8 | 20.00 | 160.00 |
| 3/16/2020 | ProductC | 9 | 30.00 | 270.00 |
| 3/8/2020 | ProductA | 5 | 10.00 | 50.00 |
| 3/10/2020 | ProductB | 3 | 20.00 | 60.00 |
| 3/13/2020 | ProductC | 1 | 30.00 | 30.00 |
| 3/12/2020 | ProductA | 7 | 10.00 | 70.00 |
| 3/10/2020 | ProductB | 7 | 20.00 | 140.00 |
| 3/24/2020 | ProductC | 9 | 30.00 | 270.00 |
| 3/15/2020 | ProductA | 8 | 10.00 | 80.00 |
| 3/10/2020 | ProductB | 5 | 20.00 | 100.00 |
| 3/10/2020 | ProductC | 4 | 30.00 | 120.00 |

**Transformation:**

The following transformation computes the third highest date in the `Date` column:

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | kthlargestdate(Date, 3) |
| Parameter: New column name | 'kthlargestdate' |

This transformation computes the third highest unique value in the `Date` column:

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | kthlargestuniquedate(Date, 3) |
| Parameter: New column name | 'kthlargestuniquedate' |

Following transformation calculates the 3rd highest date value when the OrderValue > 200:

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | kthlargestdateif(Date, 3, OrderValue > 200) |
| Parameter: New column name | 'kthlargestdateif' |

Following transformation calculates the 3rd highest unique date value when the OrderValue > 200:

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | kthlargestuniquedateif(Date, 3, OrderValue > 200) |
| Parameter: New column name | 'kthlargestuniquedateif' |

**Results:**

| Date | Product | Units | UnitCost | OrderValue | kthlargestdate | kthlargestuniquedate | kthlargestdateif | kthlargestu |
|---|---|---|---|---|---|---|---|---|
| 3/28 /2020 | ProductA | 4 | 10.00 | 40.00 | 03-24-2020 | 03-23-2020 | 03-23-2020 | 03-23-2020 |
| 3/8 /2020 | ProductB | 4 | 20.00 | 80.00 | 03-24-2020 | 03-23-2020 | 03-23-2020 | 03-23-2020 |
| 3/12 /2020 | ProductC | 2 | 30.00 | 60.00 | 03-24-2020 | 03-23-2020 | 03-23-2020 | 03-23-2020 |
| 3/23 /2020 | ProductA | 1 | 10.00 | 10.00 | 03-24-2020 | 03-23-2020 | 03-23-2020 | 03-23-2020 |
| 3/20 /2020 | ProductB | 2 | 20.00 | 40.00 | 03-24-2020 | 03-23-2020 | 03-23-2020 | 03-23-2020 |
| 3/12 /2020 | ProductC | 9 | 30.00 | 270.00 | 03-24-2020 | 03-23-2020 | 03-23-2020 | 03-23-2020 |
| 3/28 | ProductA | 5 | 10.00 | 50.00 | 03-24-2020 | 03-23-2020 | 03-23-2020 | 03-23-2020 |

| /2020 | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 3/23 /2020 | ProductB | 8 | 20.00 | 160.00 | 03-24-2020 | 03-23-2020 | 03-23-2020 | 03-23-2020 |
| 3/16 /2020 | ProductC | 9 | 30.00 | 270.00 | 03-24-2020 | 03-23-2020 | 03-23-2020 | 03-23-2020 |
| 3/8 /2020 | ProductA | 5 | 10.00 | 50.00 | 03-24-2020 | 03-23-2020 | 03-23-2020 | 03-23-2020 |
| 3/10 /2020 | ProductB | 3 | 20.00 | 60.00 | 03-24-2020 | 03-23-2020 | 03-23-2020 | 03-23-2020 |
| 3/13 /2020 | ProductC | 1 | 30.00 | 30.00 | 03-24-2020 | 03-23-2020 | 03-23-2020 | 03-23-2020 |
| 3/12 /2020 | ProductA | 7 | 10.00 | 70.00 | 03-24-2020 | 03-23-2020 | 03-23-2020 | 03-23-2020 |
| 3/10 /2020 | ProductB | 7 | 20.00 | 140.00 | 03-24-2020 | 03-23-2020 | 03-23-2020 | 03-23-2020 |
| 3/24 /2020 | ProductC | 9 | 30.00 | 270.00 | 03-24-2020 | 03-23-2020 | 03-23-2020 | 03-23-2020 |
| 3/15 /2020 | ProductA | 8 | 10.00 | 80.00 | 03-24-2020 | 03-23-2020 | 03-23-2020 | 03-23-2020 |
| 3/10 /2020 | ProductB | 5 | 20.00 | 100.00 | 03-24-2020 | 03-23-2020 | 03-23-2020 | 03-23-2020 |
| 3/10 /2020 | ProductC | 4 | 30.00 | 120.00 | 03-24-2020 | 03-23-2020 | 03-23-2020 | 03-23-2020 |

# EXAMPLE - KTHLARGEST Function

This example explores how you can use aggregation functions to calculate rank of values in a column.

**Functions**:

| Item | Description |
|---|---|
| KTHLARGEST Function | Extracts the ranked value from the values in a column, where $k=1$ returns the maximum value. The value for $k$ must be between 1 and 1000, inclusive. Inputs can be Integer, Decimal, or Datetime. |
| KTHLARGESTUNIQUE Function | Extracts the ranked unique value from the values in a column, where $k=1$ returns the maximum value. The value for $k$ must be between 1 and 1000, inclusive. Inputs can be Integer, Decimal, or Datetime. |

**Source:**

You have a set of student test scores:

| Student | Score |
|---|---|
| Anna | 84 |
| Ben | 71 |
| Caleb | 76 |
| Danielle | 87 |
| Evan | 85 |
| Faith | 92 |
| Gabe | 87 |
| Hannah | 99 |
| Ian | 73 |
| Jane | 68 |

**Transformation:**

You can use the following transformations to extract the 1st through 4th-ranked scores on the test:

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | KTHLARGEST(Score, 1) |
| Parameter: New column name | '1st' |

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | KTHLARGEST(Score, 2) |
| Parameter: New column name | '2nd' |

| Transformation Name | New formula |
|---|---|

| Parameter: Formula type | Single row formula |
|---|---|
| Parameter: Formula | KTHLARGEST(Score, 3) |
| Parameter: New column name | '3rd' |

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | KTHLARGEST(Score, 4) |
| Parameter: New column name | '4th' |

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | KTHLARGESTUNIQUE(Score, 3) |
| Parameter: New column name | '3rdUnique' |

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | KTHLARGESTUNIQUE(Score, 4) |
| Parameter: New column name | '4thUnique' |

**Results:**

When you reorganize the columns, the dataset might look like the following:

| Student | Score | 1st | 2nd | 3rd | 4th | 3rdUnique | 4thUnique |
|---|---|---|---|---|---|---|---|
| Anna | 84 | 99 | 92 | 87 | 87 | 87 | 85 |
| Ben | 71 | 99 | 92 | 87 | 87 | 87 | 85 |
| Caleb | 76 | 99 | 92 | 87 | 87 | 87 | 85 |
| Danielle | 87 | 99 | 92 | 87 | 87 | 87 | 85 |
| Evan | 85 | 99 | 92 | 87 | 87 | 87 | 85 |
| Faith | 92 | 99 | 92 | 87 | 87 | 87 | 85 |
| Gabe | 87 | 99 | 92 | 87 | 87 | 87 | 85 |
| Hannah | 99 | 99 | 92 | 87 | 87 | 87 | 85 |
| Ian | 73 | 99 | 92 | 87 | 87 | 87 | 85 |
| Jane | 68 | 99 | 92 | 87 | 87 | 87 | 85 |

Notes:

- The value 87 is both the third and fourth scores.
  - For the KTHLARGEST function, it is the output for the third and fourth ranking.
  - For the KTHLARGESTUNIQUE function, it is the output for the third ranking only.

# EXAMPLE - KTHLARGESTIF Function

This example illustrates how to use the conditional ranking functions.

**Functions:**

| Item | Description |
|------|-------------|
| KTHLARGESTIF Function | Extracts the ranked value from the values in a column, where $k=1$ returns the maximum value, when a specified condition is met. The value for $k$ must be between 1 and 1000, inclusive. Inputs can be Integer, Decimal, or Datetime. |
| KTHLARGESTUNIQUEIF Function | Extracts the ranked unique value from the values in a column, where $k=1$ returns the maximum value, when a specified condition is met. The value for $k$ must be between 1 and 1000, inclusive. Inputs can be Integer, Decimal, or Datetime. |

**Source:**

Here is some example weather data:

| date | city | rain_cm | temp_C | wind_mph |
|------|------|---------|--------|----------|
| 1/23/17 | Valleyville | 0.00 | 12.8 | 8.8 |
| 1/23/17 | Center Town | 0.31 | 9.4 | 5.3 |
| 1/23/17 | Magic Mountain | 0.00 | 0.0 | 7.3 |
| 1/24/17 | Valleyville | 0.25 | 17.2 | 3.3 |
| 1/24/17 | Center Town | 0.54 | 1.1 | 7.6 |
| 1/24/17 | Magic Mountain | 0.32 | 5.0 | 8.8 |
| 1/25/17 | Valleyville | 0.02 | 3.3 | 6.8 |
| 1/25/17 | Center Town | 0.83 | 3.3 | 5.1 |
| 1/25/17 | Magic Mountain | 0.59 | -1.7 | 6.4 |
| 1/26/17 | Valleyville | 1.08 | 15.0 | 4.2 |
| 1/26/17 | Center Town | 0.96 | 6.1 | 7.6 |
| 1/26/17 | Magic Mountain | 0.77 | -3.9 | 3.0 |
| 1/27/17 | Valleyville | 1.00 | 7.2 | 2.8 |
| 1/27/17 | Center Town | 1.32 | 20.0 | 0.2 |
| 1/27/17 | Magic Mountain | 0.77 | 5.6 | 5.2 |
| 1/28/17 | Valleyville | 0.12 | -6.1 | 5.1 |
| 1/28/17 | Center Town | 0.14 | 5.0 | 4.9 |
| 1/28/17 | Magic Mountain | 1.50 | 1.1 | 0.4 |
| 1/29/17 | Valleyville | 0.36 | 13.3 | 7.3 |
| 1/29/17 | Center Town | 0.75 | 6.1 | 9.0 |
| 1/29/17 | Magic Mountain | 0.60 | 3.3 | 6.0 |

**Transformation:**

In this case, you want to find out the second-most measures for rain, temperature, and wind in Center Town for the week.

| Transformation Name | Pivot columns |
|---|---|
| Parameter: Values | KTHLARGESTIF(rain_cm,2,city == 'Center Town') |
| Parameter: Max number of columns to create | 1 |

You can see in the preview that the value is `1.32`. Before adding it to your recipe, you change the step to the following:

| Transformation Name | Pivot columns |
|---|---|
| Parameter: Values | KTHLARGESTIF(temp_C,2,city == 'Center Town') |
| Parameter: Max number of columns to create | 1 |

The value is `20`.

For wind, you modify it to be the following, capturing the third-ranked value:

| Transformation Name | Pivot columns |
|---|---|
| Parameter: Values | KTHLARGESTIF(wind_mph,3,city == 'Center Town') |
| Parameter: Max number of columns to create | 1 |

In the results, you notice that there are two values for `8.8`. So you change the function to use the `KTHLARGESTUNIQUEIF` function instead:

| Transformation Name | Pivot columns |
|---|---|
| Parameter: Values | KTHLARGESTUNIQUEIF(wind_mph,3,city == 'Center Town') |
| Parameter: Max number of columns to create | 1 |

The result value is `7.6`. Note that this value appears twice, so if you change the rank parameter in the above transformation to `4`, the results would return a different unique ranked value (`7.3`).

**Results:**

You can choose to add any of these steps to generate an aggregated result. As an alternative, you can use a `derive` transform to insert these calculated results into new columns.

# EXAMPLE - LIST and UNIQUE Function

This example demonstrates you to extract values from one column of an array into a new column.

**Functions**:

| Item | Description |
|------|-------------|
| LIST Function | Extracts the set of values from a column into an array stored in a new column. This function is typically part of an aggregation. |
| UNIQUE Function | Extracts the set of unique values from a column into an array stored in a new column. This function is typically part of an aggregation. |
| DATEFORMAT Function | Formats a specified Datetime set of values according to the specified date format. Source values can be a reference to a column containing Datetime values. |

You have the following set of orders for two months, and you are interested in identifying the set of colors that have been sold for each product for each month and the total quantity of product sold for each month.

**Source:**

| OrderId | Date | Item | Qty | Color |
|---------|------|------|-----|-------|
| 1001 | 1/15/15 | Pants | 1 | red |
| 1002 | 1/15/15 | Shirt | 2 | green |
| 1003 | 1/15/15 | Hat | 3 | blue |
| 1004 | 1/16/15 | Shirt | 4 | yellow |
| 1005 | 1/16/15 | Hat | 5 | red |
| 1006 | 1/20/15 | Pants | 6 | green |
| 1007 | 1/15/15 | Hat | 7 | blue |
| 1008 | 4/15/15 | Shirt | 8 | yellow |
| 1009 | 4/15/15 | Shoes | 9 | brown |
| 1010 | 4/16/15 | Pants | 1 | red |
| 1011 | 4/16/15 | Hat | 2 | green |
| 1012 | 4/16/15 | Shirt | 3 | blue |
| 1013 | 4/20/15 | Shoes | 4 | black |
| 1014 | 4/20/15 | Hat | 5 | blue |
| 1015 | 4/20/15 | Pants | 6 | black |

**Transformation:**

To track by month, you need a column containing the month value extracted from the date:

| | |
|---|---|
| **Transformation Name** | `Edit column with formula` |
| **Parameter: Columns** | `Date` |
| **Parameter: Formula** | `DATEFORMAT(Date, 'MMM yyyy')` |

You can use the following transform to check the list of unique values among the colors:

| Transformation Name | Pivot columns |
|---|---|
| Parameter: Row labels | Date |
| Parameter: Values | unique(Color, 1000) |
| Parameter: Max number of columns to create | 10 |

| Date | unique_Color |
|---|---|
| Jan 2015 | ["green","blue","red","yellow"] |
| Apr 2015 | ["brown","blue","red","yellow","black","green"] |

Delete the above transform.

You can aggregate the data in your dataset, grouped by the reformatted `Date` values, and apply the `LIST` function to the `Color` column. In the same aggregation, you can include a summation function for the `Qty` column:

| Transformation Name | Pivot columns |
|---|---|
| Parameter: Row labels | Date |
| Parameter: Values | list(Color, 1000),sum(Qty) |
| Parameter: Max number of columns to create | 10 |

**Results:**

| Date | list_Color | sum_Qty |
|---|---|---|
| Jan 2015 | ["green","blue","blue","red","green","red","yellow"] | 28 |
| Apr 2015 | ["brown","blue","red","yellow","black","blue","black","green"] | 38 |

# EXAMPLE - LISTIF Functions

This example illustrates you to identify and list all values within a group that meet a specified condition.

**Functions**:

| Item | Description |
|------|-------------|
| ANYIF Function | Selects a single non-null value from rows in each group that meet a specific condition. |
| LISTIF Function | Returns list of all values in a column for rows that match a specified condition. |
| WEEKDAY Function | Derives the numeric value for the day of the week ($1$, $2$, etc.). Input must be a reference to a column containing Datetime values. |

**Source:**

The following data identifies sales figures by salespeople for a week:

| EmployeeId | Date | Sales |
|------------|------|-------|
| S001 | 1/23/17 | 25 |
| S002 | 1/23/17 | 40 |
| S003 | 1/23/17 | 48 |
| S001 | 1/24/17 | 81 |
| S002 | 1/24/17 | 11 |
| S003 | 1/24/17 | 25 |
| S001 | 1/25/17 | 9 |
| S002 | 1/25/17 | 40 |
| S003 | 1/25/17 | |
| S001 | 1/26/17 | 77 |
| S002 | 1/26/17 | 83 |
| S003 | 1/26/17 | |
| S001 | 1/27/17 | 17 |
| S002 | 1/27/17 | 71 |
| S003 | 1/27/17 | 29 |
| S001 | 1/28/17 | |
| S002 | 1/28/17 | |
| S003 | 1/28/17 | 14 |
| S001 | 1/29/17 | 2 |
| S002 | 1/29/17 | 7 |
| S003 | 1/29/17 | 99 |

**Transformation:**

In this example, you are interested in the high performers. A good day in sales is one in which an individual sells more than 80 units. First, you want to identify the day of week:

| Transformation Name | New formula |
| --- | --- |
| Parameter: Formula type | Single row formula |
| Parameter: Formula | WEEKDAY(Date) |
| Parameter: New column name | 'DayOfWeek' |

Values greater than 5 in `DayOfWeek` are weekend dates. You can use the following to identify if anyone reached this highwater marker during the workweek (non-weekend):

| Transformation Name | Pivot columns |
| --- | --- |
| Parameter: Rows labels | EmployeeId,Date |
| Parameter: Values | ANYIF(Sales, (Sales > 80 && DayOfWeek < 6)) |
| Parameter: Max number of columns to create | 1 |

Before adding the step to the recipe, you take note of the individuals who reached this mark in the `anyif_Sales` column for special recognition.

Now, you want to find out sales for individuals during the week. You can use the following to filter the data to show only for weekdays:

| Transformation Name | Pivot columns |
| --- | --- |
| Parameter: Rows labels | EmployeeId,Date |
| Parameter: Values | LISTIF(Sales, 1000, (DayOfWeek < 6)) |
| Parameter: Max number of columns to create | 1 |

To clean up, you might select and replace the following values in the listif_Sales column with empty strings:

```
["
"]
[]
```

**Results:**

| EmployeeId | Date | listif_Sales |
| --- | --- | --- |
| S001 | 1/23/17 | 25 |
| S002 | 1/23/17 | 40 |
| S003 | 1/23/17 | 48 |
| S001 | 1/24/17 | 81 |
| S002 | 1/24/17 | 11 |
| S003 | 1/24/17 | 25 |
| S001 | 1/25/17 | 40 |
| S002 | 1/25/17 | |

| | | |
|---|---|---|
| S003 | 1/25/17 | 66 |
| S001 | 1/26/17 | 77 |
| S002 | 1/26/17 | 83 |
| S003 | 1/26/17 | |
| S001 | 1/27/17 | 17 |
| S002 | 1/27/17 | 71 |
| S003 | 1/27/17 | 29 |
| S001 | 1/28/17 | |
| S002 | 1/28/17 | |
| S003 | 1/28/17 | |
| S001 | 1/29/17 | |
| S002 | 1/29/17 | |
| S003 | 1/29/17 | |

# EXAMPLE - LIST Math Functions

This example describes how to generate random array (list) data and then to apply statistical functions specifically created for arrays.

**Functions:**

| Item | Description |
|------|-------------|
| LISTSUM Function | Computes the sum of all numeric values found in input array. Input can be an array literal, a column of arrays, or a function returning an array. Input values must be of Integer or Decimal type. |
| LISTMIN Function | Computes the minimum of all numeric values found in input array. Input can be an array literal, a column of arrays, or a function returning an array. Input values must be of Integer or Decimal type. |
| LISTMAX Function | Computes the maximum of all numeric values found in input array. Input can be an array literal, a column of arrays, or a function returning an array. Input values must be of Integer or Decimal type. |
| LISTAVERAGE Function | Computes the average of all numeric values found in input array. Input can be an array literal, a column of arrays, or a function returning an array. Input values must be of Integer or Decimal type. |
| LISTVAR Function | Computes the variance of all numeric values found in input array. Input can be an array literal, a column of arrays, or a function returning an array. Input values must be of Integer or Decimal type. |
| LISTSTDEV Function | Computes the standard deviation of all numeric values found in input array. Input can be an array literal, a column of arrays, or a function returning an array. Input values must be of Integer or Decimal type. |
| LISTMODE Function | Computes the most common value of all numeric values found in input array. Input can be an array literal, a column of arrays, or a function returning an array. Input values must be of Integer or Decimal type. |

Also:

| Item | Description |
|------|-------------|
| RANGE Function | Computes an array of integers, from a beginning integer to an end (stop) integer, stepping by a third parameter. |
| RAND Function | The RAND function generates a random real number between 0 and 1. The function accepts an optional integer parameter, which causes the same set of random numbers to be generated with each job execution. |
| ROUND Function | Rounds input value to the nearest integer. Input can be an Integer, a Decimal, a column reference, or an expression. Optional second argument can be used to specify the number of digits to which to round. |

**Source:**

For this example, you can generate some randomized data using the following steps. First, you need to seed an array with a range of values using the RANGE function:

| | |
|---|---|
| **Transformation Name** | `New formula` |
| **Parameter: Formula type** | `Single row formula` |
| **Parameter: Formula** | `RANGE(5, 50, 5)` |
| **Parameter: New column name** | `'myArray1'` |

Then, unpack this array, so you can add a random factor:

| | |
|---|---|
| **Transformation Name** | `Unnest Objects into columns` |
| **Parameter: Column** | `myArray1` |
| **Parameter: Paths to elements** | `'[0]', '[1]', '[2]', '[3]', '[4]', '[5]', '[6]', '[7]', '[8]', '[9]'` |

| Parameter: Remove elements from original | true |
|---|---|
| Parameter: Include original column name | true |

Add the randomizing factor. Here, you are adding randomization around individual values:  x-1 < x < x+4.

| Transformation Name | Edit column with formula |
|---|---|
| Parameter: Columns | myArray1_0~myArray1_8 |
| Parameter: Formula | IF(RAND() > 0.5, $col + (5 * RAND()), $col - RAND()) |

To make the numbers easier to manipulate, you can round them to two decimal places:

| Transformation Name | Edit column with formula |
|---|---|
| Parameter: Columns | myArray1_0~myArray1_8 |
| Parameter: Formula | ROUND($col, 2) |

Renest these columns into an array:

| Transformation Name | Nest columns into Objects |
|---|---|
| Parameter: Columns | myArray1_0, myArray1_1, myArray1_2, myArray1_3, myArray1_4, myArray1_5, myArray1_6, myArray1_7, myArray1_8 |
| Parameter: Nest columns to | Array |
| Parameter: New column name | 'myArray2' |

Delete the unused columns:

| Transformation Name | Delete columns |
|---|---|
| Parameter: Columns | myArray1_0~myArray1_8,myArray1 |
| Parameter: Action | Delete selected columns |

Your data should look similar to the following:

| myArray2 |
|---|
| ["8.29","9.63","14.63","19.63","24.63","29.63","34.63","39.63","44.63"] |
| ["8.32","14.01","19.01","24.01","29.01","34.01","39.01","44.01","49.01"] |
| ["4.55","9.58","14.58","19.58","24.58","29.58","34.58","39.58","44.58"] |
| ["9.22","14.84","19.84","24.84","29.84","34.84","39.84","44.84","49.84"] |
| ["8.75","13.36","18.36","23.36","28.36","33.36","38.36","43.36","48.36"] |
| ["8.47","14.76","19.76","24.76","29.76","34.76","39.76","44.76","49.76"] |
| ["4.93","9.99","14.99","19.99","24.99","29.99","34.99","39.99","44.99"] |
| ["4.65","14.98","19.98","24.98","29.98","34.98","39.98","44.98","49.98"] |

["7.80","14.62","19.62","24.62","29.62","34.62","39.62","44.62","49.62"]

["9.32","9.96","14.96","19.96","24.96","29.96","34.96","39.96","44.96"]

**Transformation:**

These steps demonstrate the individual math functions that you can apply to your list data without unnesting it:

> **NOTE:** The NUMFORMAT function has been wrapped around each list function to account for any floating-point errors or additional digits in the results.

Sum of all values in the array (list):

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | NUMFORMAT(LISTSUM(myArray2), '#.##') |
| Parameter: New column name | 'arraySum' |

Minimum of all values in the array (list):

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | NUMFORMAT(LISTMIN(myArray2), '#.##') |
| Parameter: New column name | 'arrayMin' |

Maximum of all values in the array (list):

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | NUMFORMAT(LISTMAX(myArray2), '#.##') |
| Parameter: New column name | 'arrayMax' |

Average of all values in the array (list):

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | NUMFORMAT(LISTAVERAGE(myArray2), '#.##') |
| Parameter: New column name | 'arrayAvg' |

Variance of all values in the array (list):

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | NUMFORMAT(LISTVAR(myArray2), '#.##') |
| Parameter: New column name | 'arrayVar' |

Standard deviation of all values in the array (list):

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | NUMFORMAT(LISTSTDEV(myArray2), '#.##') |
| Parameter: New column name | 'arrayStDv' |

Mode (most common value) of all values in the array (list):

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | NUMFORMAT(LISTMODE(myArray2), '#.##') |
| Parameter: New column name | 'arrayMode' |

**Results:**

Results for the first four math functions:

| myArray2 | arrayAvg | arrayMax | arrayMin | arraySum |
|---|---|---|---|---|
| ["8.29","9.63","14.63","19.63","24.63","29.63","34.63","39.63","44.63"] | 25.04 | 44.63 | 8.29 | 225.33 |
| ["8.32","14.01","19.01","24.01","29.01","34.01","39.01","44.01","49.01"] | 28.93 | 49.01 | 8.32 | 260.4 |
| ["4.55","9.58","14.58","19.58","24.58","29.58","34.58","39.58","44.58"] | 24.58 | 44.58 | 4.55 | 221.19 |
| ["9.22","14.84","19.84","24.84","29.84","34.84","39.84","44.84","49.84"] | 29.77 | 49.84 | 9.22 | 267.94 |
| ["8.75","13.36","18.36","23.36","28.36","33.36","38.36","43.36","48.36"] | 28.4 | 48.36 | 8.75 | 255.63 |
| ["8.47","14.76","19.76","24.76","29.76","34.76","39.76","44.76","49.76"] | 29.62 | 49.76 | 8.47 | 266.55 |
| ["4.93","9.99","14.99","19.99","24.99","29.99","34.99","39.99","44.99"] | 24.98 | 44.99 | 4.93 | 224.85 |
| ["4.65","14.98","19.98","24.98","29.98","34.98","39.98","44.98","49.98"] | 29.39 | 49.98 | 4.65 | 264.49 |
| ["7.80","14.62","19.62","24.62","29.62","34.62","39.62","44.62","49.62"] | 29.42 | 49.62 | 7.8 | 264.76 |
| ["9.32","9.96","14.96","19.96","24.96","29.96","34.96","39.96","44.96"] | 25.44 | 44.96 | 9.32 | 229 |

Results for the statistical functions:

| myArray2 | arrayMode | arrayStDv | arrayVar |
|---|---|---|---|
| ["8.29","9.63","14.63","19.63","24.63","29.63","34.63","39.63","44.63"] | | 12.32 | 151.72 |

| | | |
|---|---|---|
| ["8.32","14.01","19.01","24.01","29.01","34.01","39.01","44.01","49.01"] | 13.03 | 169.78 |
| ["4.55","9.58","14.58","19.58","24.58","29.58","34.58","39.58","44.58"] | 12.92 | 166.8 |
| ["9.22","14.84","19.84","24.84","29.84","34.84","39.84","44.84","49.84"] | 13.02 | 169.46 |
| ["8.75","13.36","18.36","23.36","28.36","33.36","38.36","43.36","48.36"] | 12.84 | 164.95 |
| ["8.47","14.76","19.76","24.76","29.76","34.76","39.76","44.76","49.76"] | 13.14 | 172.56 |
| ["4.93","9.99","14.99","19.99","24.99","29.99","34.99","39.99","44.99"] | 12.92 | 166.93 |
| ["4.65","14.98","19.98","24.98","29.98","34.98","39.98","44.98","49.98"] | 13.9 | 193.16 |
| ["7.80","14.62","19.62","24.62","29.62","34.62","39.62","44.62","49.62"] | 13.23 | 175.08 |
| ["9.32","9.96","14.96","19.96","24.96","29.96","34.96","39.96","44.96"] | 12.21 | 149.17 |

Since all values are unique within an individual array, there is no most common value in any of them, which yields empty values for the `arrayMode` column.

# EXAMPLE - Logical Functions

This example demonstrate the `AND`, `OR`, and `NOT` logical functions.

In this example, the dataset contains results from survey data on two questions about customers. The yes/no answers to each question determine if the customer is 1) still active, and 2) interested in a new offering.

**Functions:**

| Item | Description |
|------|-------------|
| AND Function | Returns `true` if both arguments evaluate to `true`. Equivalent to the `&&` operator. |
| OR Function | Returns `true` if either argument evaluates to `true`. Equivalent to the `||` operator. |
| NOT Function | Returns `true` if the argument evaluates to `false`, and vice-versa. Equivalent to the `!` operator. |

**Source:**

| Customer | isActive | isInterested |
|----------|----------|--------------|
| CustA | Y | Y |
| CustB | Y | N |
| CustC | N | Y |
| CustD | N | N |

**Transformation:**

Customers that are both active and interested should receive a phone call:

| Transformation Name | New formula |
|---------------------|-------------|
| **Parameter: Formula type** | `Single row formula` |
| **Parameter: Formula** | `AND(isActive, isInterested)` |
| **Parameter: New column name** | `'phoneCall'` |

Customers that are either active or interested should receive an email:

| Transformation Name | New formula |
|---------------------|-------------|
| **Parameter: Formula type** | `Single row formula` |
| **Parameter: Formula** | `OR(isActive, isInterested)` |
| **Parameter: New column name** | `'sendEmail'` |

Customers that are neither active or interested should be dropped from consideration for the offering:

| Transformation Name | New formula |
|---------------------|-------------|
| **Parameter: Formula type** | `Single row formula` |
| **Parameter: Formula** | `AND(NOT(isActive),NOT(isInterested)` |

| | |
|---|---|
| **Parameter: New column name** | `'dropCust'` |

A savvy marketer might decide that if a customer receives a phone call, that customer should not be bothered with an email, as well:

| | |
|---|---|
| **Transformation Name** | `Edit column with formula` |
| **Parameter: Columns** | `sendEmail` |
| **Parameter: Formula** | `IF(phoneCall == "TRUE", FALSE, sendEmail)` |

**Results:**

| Customer | isActive | isInterested | dropCust | sendEmail | phoneCall |
|---|---|---|---|---|---|
| CustA | Y | Y | FALSE | FALSE | TRUE |
| CustB | Y | N | FALSE | TRUE | FALSE |
| CustC | N | Y | FALSE | TRUE | FALSE |
| CustD | N | N | TRUE | FALSE | FALSE |

# EXAMPLE - Nest and Unnest JSON Records

This example illustrates how you can use unnesting and nesting transformations to reshape your JSON data.

The following example contains records of images from a website:

```
{"metrics": [{"rank": "1043", "score" : "9679"}], "caption": "Such a good boy!", "id": "9kt8ex", "url":
"https://www.example.com/w285fpp11.jpg", "filename": "w285fpp11.jpg"}
{"metrics": [{"rank": "1042", "score" : "9681"}], "caption": "This sweet puppy has transformed our life!",
"id": "9x2774", "url": "https://www.example.com/fmll0cy11.jpg", "filename": "fmll0cy11.jpg"}
{"metrics": [{"rank": "1041", "score" : "9683"}], "caption": "We sure love our fur babies.", "id": "a8guou",
"url": "https://www.example.com/mljnmq521.jpg", "filename": "mljnmq521.jpg"}
```

**Steps**:

1. Add the JSON-based imported dataset to a flow and create a recipe for it.
2. Select the recipe, and click **Edit Recipe...**.

In the Transformer page, the example above should look like the following:

| metrics | caption | id | url | filename |
|---|---|---|---|---|
| [{"rank":"1043","score":"9679"}] | Such a good boy! | 9kt8ex | https://www.example.com/w285fpp11.jpg | w285fpp11.jpg |
| [{"rank":"1042","score":"9681"}] | This sweet puppy has transformed our life! | 9x2774 | https://www.example.com/fmll0cy11.jpg | fmll0cy11.jpg |
| [{"rank":"1041","score":"9683"}] | We sure love our fur babies. | a8guou | https://www.example.com/mljnmq521.jpg | mljnmq521.jpg |

Suppose you want to nest the `url` and `filename` columns into a nested array called, `resources`.

**Steps:**

1. SHIFT + click the `url` and `filename` columns. Then, select **Nest columns** in the right-hand panel. This transformation should look like the following:

| | |
|---|---|
| **Transformation Name** | Nest columns into Objects |
| **Parameter: column1** | url |
| **Parameter: column2** | filename |
| **Parameter: Nest columns to** | Object |
| **Parameter: New column name** | column1 |

2. `column1` now contains an Object mapping of the two columns. You can now nest this column again into an Array:

| | |
|---|---|
| **Transformation Name** | Nest columns into Objects |
| **Parameter: Columns** | column1 |
| **Parameter: Nest columns to** | Array |
| **Parameter: New column name** | resources |

3. Delete `column1`.
4. Continue nesting other columns in a similar fashion. Repeat the above steps for the next level of the hierarchy in your dataset.

# EXAMPLE - Nest Columns into Arrays

This section provides simple examples of nesting columns into Arrays by extracting values from a column or nesting one or more columns into an Array column.

**Create by extraction:**

You can create an array of values by extracting pattern-based values from a specified column. The following transformation extracts from the `msg` column a list of all values where all letters are capitalized and places them into the new `acronyms` column:

| Transformation Name | Extract matches into Array |
|---|---|
| **Parameter: Column** | `msg` |
| **Parameter: Pattern matching elements in the list** | `` `{upper}+` `` |
| **Parameter: New column name** | `acronyms` |

| msg | acronyms |
|---|---|
| SCUBA, IMHO, is the greatest sport in the world. | ["SCUBA","IMHO"] |
| | [] |
| LOL, that assignment you finished is DOA. You need to fix it PDQ. | ["LOL","DOA","Y","PDQ"] |

Notes:

- An empty input column value renders an empty array.
- In the final row, the  Pattern  matches on the `"Y"` value. To fix this, you can change the Pattern matching value to the following, which matches on two or more uppercase letters in a row:

```
`{upper}{upper}+`
```

**Create by nesting:**

You can create arrays by nesting together the values from multiple columns:

| num1 | num2 | num3 |
|---|---|---|
| 11 | 12 | 13 |
| 14 | 15 | 16 |
| 17 | 18 | 19 |

You can nest the values in `num1` and `num2` into a single array and then to nest the array with `num3`:

> **NOTE:** If you are nesting a multi-level array, you should nest from the lowest level to the top level.

| Transformation Name | Nest columns into Objects |
|---|---|
| **Parameter: Columns1** | `num1` |
| **Parameter: Columns2** | `num2` |
| **Parameter: Nest columns to** | `Array` |

| | |
|---|---|
| **Parameter: New column name** | nest1 |

Then, you can perform the nesting of the top-level elements:

> **NOTE:** The order in which you list the columns to nest determines the order in which the elements appear in the generated array.

| | |
|---|---|
| **Transformation Name** | Nest columns into Objects |
| **Parameter: Columns1** | nest1 |
| **Parameter: Columns2** | num3 |
| **Parameter: Nest columns to** | Array |
| **Parameter: New column name** | nest2 |

In the generated columns, you notice that all values are quoted, even though these values are integers.

> **NOTE:** Elements that are generated into arrays using a nest transformation are always rendered as quoted values.

You can use the following transformation to remove the quotes from the `nest2` column:

| | |
|---|---|
| **Transformation Name** | Replace text or patterns |
| **Parameter: Column** | nest2 |
| **Parameter: Find** | '"' |
| **Parameter: Replace** | (empty) |
| **Parameter: Match all occurrences** | true |

| num1 | num2 | num3 | nest2 |
|---|---|---|---|
| 11 | 12 | 13 | [[11,12],13] |
| 14 | 15 | 16 | [[14,15],16] |
| 17 | 18 | 19 | [[17,18],19] |

# EXAMPLE - Nest Columns into Objects

This section provides a simple example of nesting columns into a new column of Object data type.

**Source:**

In the following example, furniture product dimensions are stored in separate columns in cm.

| Category | Name | Length_cm | Width_cm | Height_cm |
|----------|------|-----------|----------|-----------|
| bench | Hooska | 118.11 | 74.93 | 46.34 |
| lamp | Tansk | 30.48 | 30.48 | 165.1 |
| bookshelf | Brock | 27.94 | 160.02 | 201.93 |
| couch | Loafy | 95 | 227 | 83 |

**Transformation:**

Use the `nest` transform to bundle the data into a single column.

| Transformation Name | Nest columns into Objects |
|---------------------|---------------------------|
| **Parameter: Columns** | `Length_cm,Width_cm,Height_cm` |
| **Parameter: Nest columns to** | `Object` |
| **Parameter: New column name** | `'Dimensions'` |

**Results:**

| Category | Name | Length_cm | Width_cm | Height_cm | Dimensions |
|----------|------|-----------|----------|-----------|------------|
| bench | Hooska | 118.11 | 74.93 | 46.34 | {"Length_cm":"118.11","Width_cm":"74.93","Height_cm":"46.34"} |
| lamp | Tansk | 30.48 | 30.48 | 165.1 | {"Length_cm":"30.48","Width_cm":"30.48","Height_cm":"165.1"} |
| bookshelf | Brock | 27.94 | 160.02 | 201.93 | {"Length_cm":"27.94","Width_cm":"160.02","Height_cm":"201.93"} |
| couch | Loafy | 95 | 227 | 83 | {"Length_cm":"95","Width_cm":"227","Height_cm":"83"} |

# EXAMPLE - Nested Functions

This example illustrates how to use the nested functions.

**Functions**:

| Item | Description |
|------|-------------|
| ARRAYCONCAT Function | Combines the elements of one array with another, listing all elements of the first array before listing all elements of the second array. |
| ARRAYINTERSECT Function | Generates an array containing all elements that appear in multiple input arrays, referenced as column names or array literals. |
| ARRAYCROSS Function | Generates a nested array containing the cross-product of all elements in two or more arrays. |
| ARRAYUNIQUE Function | Generates an array of all unique elements among one or more arrays. |

**Source:**

Code formatting has been applied to improve legibility.

| Item | ArrayA | ArrayB |
|------|--------|--------|
| Item1 | `["A","B","C"]` | `["1","2","3"]` |
| Item2 | `["A","B"]` | `["A","B","C"]` |
| Item3 | `["D","E","F"]` | `["4","5","6"]` |

**Transformation:**

You can apply the following transforms in the following order. Note that the column names must be different from the transform name, which is a reserved word.

| Transformation Name | New formula |
|---------------------|-------------|
| **Parameter: Formula type** | Single row formula |
| **Parameter: Formula** | ARRAYCONCAT([Letters,Numerals]) |
| **Parameter: New column name** | 'concat2' |

| Transformation Name | New formula |
|---------------------|-------------|
| **Parameter: Formula type** | Single row formula |
| **Parameter: Formula** | ARRAYINTERSECT([Letters,Numerals]) |
| **Parameter: New column name** | 'intersection2' |

| Transformation Name | New formula |
|---------------------|-------------|
| **Parameter: Formula type** | Single row formula |
| **Parameter: Formula** | ARRAYCROSS([Letters,Numerals]) |
| **Parameter: New column** | 'cross2' |

| | |
|---|---|
| **name** | |

| | |
|---|---|
| **Transformation Name** | `New formula` |
| **Parameter: Formula type** | `Single row formula` |
| **Parameter: Formula** | `ARRAYUNIQUE([Letters,Numerals])` |
| **Parameter: New column name** | `'unique2'` |

**Results:**

For display purposes, the results table has been broken down into three separate sets of columns.

Column set 1:

| Item | ArrayA | ArrayB | concat2 | intersection2 |
|---|---|---|---|---|
| Item1 | `["A","B","C"]` | `["1","2","3"]` | `["A","B","C","1","2","3"]` | `[ ]` |
| Item2 | `["A","B"]` | `["A","B","C"]` | `["A","B","A","B","C"]` | `["A","B"]` |
| Item3 | `["D","E","F"]` | `["4","5","6"]` | `["D","E","F","4","5","6"]` | `[ ]` |

Column set 2:

| Item | cross2 |
|---|---|
| Item1 | `[["A","1"],["A","2"],["A","3"],["B","1"],["B","2"],["B","3"],["C","1"],["C","2"],["C","3"]]` |
| Item2 | `[["A","A"],["A","B"],["A","C"],["B","A"],["B","B"],["B","C"]]` |
| Item3 | `[["D","4"],["D","5"],["D","6"],["E","4"],["E","5"],["E","6"], ["F","4"],["F","5"],["F","6"]]` |

Column set 3:

| Item | unique2 |
|---|---|
| Item1 | `["A","B","C","1","2","3"]` |
| Item2 | `["A","B","C"]` |
| Item3 | `["D","E","F","4","5","6"]` |

# EXAMPLE - Nest JSON Records

This section illustrates a simple example of how to nest tabular data into JSON records.

**Source:**

The following source data is a mixture of homogeneous (Score1, Score2, Score3) and heterogeneous (Height-Ins, Weight-Lbs, ShoeSize-US) columns for each row (Student).

| Student | Score1 | Score2 | Score3 | Height-Ins | Weight-Lbs | ShoeSize-US |
|---------|--------|--------|--------|------------|------------|-------------|
| StudentA | 84 | 89 | 92 | 68 | 140 | 8 |
| StudentB | 98 | 94 | 96 | 63 | 105 | 6 |
| StudentC | 81 | 83 | 82 | 65 | 114 | 7 |

**Transformation:**

You can use the following transformations to assemble the above data into a JSON object.

In this first transformation, you nest the test scores columns. Since this data has the same meaning across all three columns (homogeneous), an Array is the proper storage object:

| Transformation Name | Nest columns into Objects |
|---------------------|---------------------------|
| **Parameter: Columns1** | Score1 |
| **Parameter: Columns2** | Score2 |
| **Parameter: Columns3** | Score3 |
| **Parameter: Nest columns to** | Array |
| **Parameter: New column name** | scores |

Then, you can delete the three source columns, yielding the following:

| Student | scores | Height-Ins | Weight-Lbs | ShoeSize-US |
|---------|--------|------------|------------|-------------|
| StudentA | ["84","89","92"] | 68 | 140 | 8 |
| StudentB | ["98","94","96"] | 63 | 105 | 6 |
| StudentC | ["81","83","82"] | 65 | 114 | 7 |

Next, you can nest the other three data columns. Since the data has different meanings in each column, you should carry forward the column names as keys in the nested data, which should be an Object type:

| Transformation Name | Nest columns into Objects |
|---------------------|---------------------------|
| **Parameter: Columns** | {Height-Ins},{Weight-Lbs},{ShoeSize-Us} |
| **Parameter: Nest columns to** | Object |
| **Parameter: New column name** | 'measurements' |

After you delete the source columns, you should have the following:

| Student | scores | measurements |
|---------|--------|--------------|

| | | |
|---|---|---|
| StudentA | ["84","89","92"] | {"Height-Ins":"68","Weight-Lbs":"140","ShoeSize-US":"8"} |
| StudentB | ["98","94","96"] | {"Height-Ins":"63","Weight-Lbs":"105","ShoeSize-US":"6"} |
| StudentC | ["81","83","82"] | {"Height-Ins":"65","Weight-Lbs":"114","ShoeSize-US":"7"} |

> **NOTE:** You may have noticed that the column names for the nested columns start with a lower-case letter, which follows JSON formatting standards. The `Student` column should be renamed to `student`.

Now that you have the detail columns nested into an array and an object, you can nest these columns into an object for each student, as in the following transformation:

| Transformation Name | Nest columns into Objects |
|---|---|
| **Parameter: Columns1** | `student` |
| **Parameter: Columns2** | `scores` |
| **Parameter: Columns3** | `measurement` |
| **Parameter: Nest columns to** | `Object` |
| **Parameter: New column name** | `column1` |

After you delete the three source columns, you can rename `column1` to `student`.

**Results:**

| student |
|---|
| {"student":"StudentA","scores":["84","89","92"],"measurements":{"ShoeSize-Us":"8","Weight-Lbs":"140","Height-Ins":"68"}} |
| {"student":"StudentB","scores":["98","94","96"],"measurements":{"ShoeSize-Us":"6","Weight-Lbs":"105","Height-Ins":"63"}} |
| {"student":"StudentC","scores":["81","83","82"],"measurements":{"ShoeSize-Us":"7","Weight-Lbs":"114","Height-Ins":"65"}} |

When you run the job, generate a JSON output to produce the above in JSONLines format.

# EXAMPLE - NEXT Function

This example covers how to use the NEXT function to create windows of data from the current row and subsequent (next) rows in the dataset. You can then apply rolling computations across these windows of data.

**Functions:**

| Item | Description |
|------|-------------|
| NEXT Function | Extracts the value from a column that is a specified number of rows after the current value. |
| ROLLINGAVE RAGE Function | Computes the rolling average of values forward or backward of the current row within the specified column. |
| NUMFORMAT Function | Formats a numeric set of values according to the specified number formatting. Source values can be a literal numeric value, a function returning a numeric value, or reference to a column containing an Integer or Decimal values. |

**Source:**

The following dataset contains order information for the preceding 12 months. You want to compare the current month's average against the preceding quarter.

| Date | Amount |
|------|--------|
| 12/31/15 | 118 |
| 11/30/15 | 6 |
| 10/31/15 | 443 |
| 9/30/15 | 785 |
| 8/31/15 | 77 |
| 7/31/15 | 606 |
| 6/30/15 | 421 |
| 5/31/15 | 763 |
| 4/30/15 | 305 |
| 3/31/15 | 824 |
| 2/28/15 | 135 |
| 1/31/15 | 523 |

**Transformation:**

Using the `ROLLINGAVERAGE` function, you can generate a column containing the rolling average of the current month and the two previous months:

| | |
|------|------|
| **Transformation Name** | `Window` |
| **Parameter: Formulas** | `ROLLINGAVERAGE(Amount, 3, 0)` |
| **Parameter: Order by** | `-Date` |

Note the sign of the second parameter and the `order` parameter. The sort is in the reverse order of the `Date` parameter, which preserves the current sort order. As a result, the second parameter, which identifies the number of rows to use in the calculation, must be positive to capture the previous months.

Technically, this computation does not capture the prior quarter, since it includes the current quarter as part of the computation. You can use the following column to capture the rolling average of the preceding month, which then becomes the true rolling average for the prior quarter. The `window` column refers to the name of the column generated from the previous step:

| | |
|---|---|
| Transformation Name | `Window` |
| Parameter: Formulas | `NEXT(window, 1)` |
| Parameter: Order by | `-Date` |

Note that the order parameter must be preserved. This new column, `window1`, contains your prior quarter rolling average:

| | |
|---|---|
| Transformation Name | `Rename columns` |
| Parameter: Option | `Manual rename` |
| Parameter: Column | `window1` |
| Parameter: New column name | `'Amount_PriorQtr'` |

You can reformat this numeric value:

| | |
|---|---|
| Transformation Name | `Edit column with formula` |
| Parameter: Columns | `Amount_PriorQtr` |
| Parameter: Formula | `NUMFORMAT(Amount_PriorQtr, '###.00')` |

You can use the following transformation to calculate the net change. This formula computes the change as a percentage of the prior quarter and then formats it as a two-digit percentage.

| | |
|---|---|
| Transformation Name | `New formula` |
| Parameter: Formula type | `Single row formula` |
| Parameter: Formula | `NUMFORMAT(((Amount - Amount_PriorQtr) / Amount_PriorQtr) * 100, '##.##')` |
| Parameter: New column name | `'NetChangePct_PriorQtr'` |

**Results:**

> **NOTE:** You might notice that there are computed values for `Amount_PriorQtr` for February and March. These values do not factor in a full three months because the data is not present. The January value does not exist since there is no data preceding it.

| Date | Amount | Amount_PriorQtr | NetChangePct_PriorQtr |
|---|---|---|---|
| 12/31/15 | 118 | 411.33 | -71.31 |
| 11/30/15 | 6 | 435.00 | -98.62 |
| 10/31/15 | 443 | 489.33 | -9.47 |
| 9/30/15 | 785 | 368.00 | 113.32 |

| | | | |
|---|---|---|---|
| 8/31/15 | 77 | 596.67 | -87.1 |
| 7/31/15 | 606 | 496.33 | 22.1 |
| 6/30/15 | 421 | 630.67 | -33.25 |
| 5/31/15 | 763 | 421.33 | 81.09 |
| 4/30/15 | 305 | 494.00 | -38.26 |
| 3/31/15 | 824 | 329.00 | 150.46 |
| 2/28/15 | 135 | 523.00 | -.74.19 |
| 1/31/15 | 523 | | |

# EXAMPLE - NOW and TODAY Functions

This example illustrates you to generate the date and time values for the current date and timestamp in the specified time zone.

**Functions**:

| Item | Description |
|---|---|
| NOW Function | Derives the timestamp for the current time in UTC time zone. You can specify a different time zone by optional parameter. |
| TODAY Function | Derives the value for the current date in UTC time zone. You can specify a different time zone by optional parameter. |
| DATEDIF Function | Calculates the difference between two valid date values for the specified units of measure. |

**Source:**

The following table includes flight arrival information for Los Angeles International airport.

| FlightNumber | Gate | Arrival |
|---|---|---|
| 1234 | 1 | 2/15/17 11:35 |
| 212 | 2 | 2/15/17 11:58 |
| 510 | 3 | 2/15/17 11:21 |
| 8401 | 4 | 2/15/17 12:08 |
| 99 | 5 | 2/16/17 12:12 |
| 116 | 6 | 2/16/17 13:32 |
| 876 | 7 | 2/15/17 16:43 |
| 9494 | 8 | 2/15/17 21:00 |
| 102 | 9 | 2/14/17 19:21 |
| 77 | 10 | 2/16/17 12:31 |

**Transformation:**

You are interested in generating a status report on today's flights. To assist, you must generate columns with the current date and time values:

> **Tip:** You should create separate columns containing static values for NOW and TODAY functions. Avoid creating multiple instances of each function in your dataset, as the values calculated in them can vary at execution time.

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | NOW('America\/Los_Angeles') |
| Parameter: New column name | 'currentTime' |

| Transformation Name | New formula |
|---|---|
| | |

| Parameter: Formula type | `Single row formula` |
|---|---|
| Parameter: Formula | `TODAY('America\/Los_Angeles')` |
| Parameter: New column name | `'currentDate'` |

Next, you want to identify the flights that are landing today. In this case, you can use the `DATEDIF` function to determine if the `Arrival` value matches the `currentTime` value within one day:

> **NOTE:** The `DATEDIF` function computes difference based on the difference from the first date to the second date based on the unit of measure. So, a timestamp that is 23 hours difference from the base timestamp can be within the same unit of day, even though the dates may be different (`2/15/2017` vs. `2/14/2017`).

| Transformation Name | `New formula` |
|---|---|
| Parameter: Formula type | `Single row formula` |
| Parameter: Formula | `DATEDIF(currentDate, Arrival, day)` |
| Parameter: New column name | `'today'` |

Since you are focusing on today only, you can remove all of the rows that do not apply to today:

| Transformation Name | `Filter rows` |
|---|---|
| Parameter: Condition | `Custom formula` |
| Parameter: Type of formula | `Custom single` |
| Parameter: Condition | `today <> 0` |
| Parameter: Action | `Delete matching rows` |

Now focusing on today's dates, you can calculate the difference between the current time and the arrival time by the minute:

| Transformation Name | `New formula` |
|---|---|
| Parameter: Formula type | `Single row formula` |
| Parameter: Formula | `DATEDIF(currentTime, Arrival, minute)` |
| Parameter: New column name | `'status'` |

Using the numeric values in the `status` column, you can compose the following transform, which identifies status of each flight:

| Transformation Name | `Edit column with formula` |
|---|---|
| Parameter: Columns | `status` |
| Parameter: Formula | `if(status < -20, 'arrived', if(status > 20, 'scheduled', if(status <= 0, 'landed', 'arriving')))` |

**Results:**

You now have a daily flight status report:

| currentDate | currentTime | FlightNumber | Gate | Arrival | status | today |
|---|---|---|---|---|---|---|
| 2017-02-15 | 2017-02-15·11:46:12 | 1234 | 1 | 2/15/17 11:35 | landed | 0 |
| 2017-02-15 | 2017-02-15·11:46:12 | 212 | 2 | 2/15/17 11:58 | arriving | 0 |
| 2017-02-15 | 2017-02-15·11:46:12 | 510 | 3 | 2/15/17 11:21 | arrived | 0 |
| 2017-02-15 | 2017-02-15·11:46:12 | 8401 | 4 | 2/15/17 12:08 | scheduled | 0 |
| 2017-02-15 | 2017-02-15·11:46:12 | 876 | 7 | 2/15/17 16:43 | scheduled | 0 |
| 2017-02-15 | 2017-02-15·11:46:12 | 9494 | 8 | 2/15/17 21:00 | scheduled | 0 |
| 2017-02-15 | 2017-02-15·11:46:12 | 102 | 9 | 2/14/17 19:21 | arrived | 0 |

The `currentDate`, `currentTime`, and `today` columns can be deleted.

# EXAMPLE - Numeric Functions

This example demonstrates how to use numeric functions to perform computations in your recipe steps.

**Functions:**

| Item | Description |
|---|---|
| ADD Function | Returns the value of summing the first argument and the second argument. Equivalent to the + operator. |
| MOD Function | Returns the modulo value, which is the remainder of dividing the first argument by the second argument. Equivalent to the % operator. |
| NEGATE Function | Returns the opposite of the value that is the first argument. Equivalent to the − operator placed in front of the argument. |
| SUBTRACT Function | Returns the value of subtracting the second argument from the first argument. Equivalent to the − operator. |
| MULTIPLY Function | Returns the value of multiplying the first argument by the second argument. Equivalent to the * operator. |
| DIVIDE Function | Returns the value of dividing the first argument by the second argument. Equivalent to the / operator. |
| LCM Function | Returns the least common multiple shared by the first and second arguments. |

**Source:**

| ValueA | ValueB |
|---|---|
| 8 | 2 |
| 10 | 4 |
| 15 | 10 |
| 5 | 6 |

**Transformation:**

Execute the following transformation steps:

| Transformation Name | New formula |
|---|---|
| **Parameter: Formula type** | Single row formula |
| **Parameter: Formula** | ADD(ValueA, ValueB) |
| **Parameter: New column name** | 'add' |

| Transformation Name | New formula |
|---|---|
| **Parameter: Formula type** | Single row formula |
| **Parameter: Formula** | SUBTRACT(ValueA, ValueB) |
| **Parameter: New column name** | 'subtract' |

| Transformation Name | New formula |
|---|---|
| **Parameter: Formula type** | Single row formula |

| Parameter: Formula | MULTIPLY(ValueA, ValueB) |
|---|---|
| Parameter: New column name | 'multiply' |

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | DIVIDE(ValueA, ValueB) |
| Parameter: New column name | 'divide' |

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | MOD(ValueA, ValueB) |
| Parameter: New column name | 'mod' |

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | NEGATE(ValueA) |
| Parameter: New column name | 'negativeA' |

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | LCM(ValueA, ValueB) |
| Parameter: New column name | 'lcm' |

**Results:**

With a bit of cleanup, your dataset results might look like the following:

| ValueA | ValueB | lcm | negativeA | mod | divide | multiply | subtract | add |
|---|---|---|---|---|---|---|---|---|
| 8 | 2 | 8 | -8 | 0 | 4 | 16 | 6 | 10 |
| 10 | 4 | 20 | -10 | 2 | 2.5 | 40 | 6 | 14 |
| 15 | 10 | 30 | -15 | 5 | 1.5 | 150 | 5 | 25 |
| 5 | 6 | 30 | -5 | 5 | 0.833333333 | 30 | -1 | 11 |

# EXAMPLE - Percentile Functions

This example illustrates you to apply percentile functions.

**Functions**:

| Item | Description |
|------|-------------|
| MEDIAN Function | Computes the median from all row values in a column or group. Input column can be of Integer or Decimal. |
| PERCENTILE Function | Computes a specified percentile across all row values in a column or group. Input column can be of Integer or Decimal. |
| QUARTILE Function | Computes a specified quartile across all row values in a column or group. Input column can be of Integer or Decimal. |
| APPROXIMATEMEDIAN Function | Computes the approximate median from all row values in a column or group. Input column can be of Integer or Decimal. |
| APPROXIMATEPERCENTILE Function | Computes an approximation for a specified percentile across all row values in a column or group. Input column can be of Integer or Decimal. |
| APPROXIMATEQUARTILE Function | Computes an approximation for a specified quartile across all row values in a column or group. Input column can be of Integer or Decimal. |

**Source:**

The following table lists each student's height in inches:

| Student | Height |
|---------|--------|
| 1 | 64 |
| 2 | 65 |
| 3 | 63 |
| 4 | 64 |
| 5 | 62 |
| 6 | 66 |
| 7 | 66 |
| 8 | 65 |
| 9 | 69 |
| 10 | 66 |
| 11 | 73 |
| 12 | 69 |
| 13 | 69 |
| 14 | 61 |
| 15 | 64 |
| 16 | 61 |
| 17 | 71 |
| 18 | 67 |
| 19 | 73 |
| 20 | 66 |

**Transformation:**

Use the following transformations to calculate the median height in inches, a specified percentile and the first quartile.

- The first function uses a precise algorithm which can be slow to execute across large datasets.
- The second function uses an appropriate approximation algorithm, which is much faster to execute across large datasets.
    - These approximate functions can use an error boundary parameter, which is set to `0.4` (0.4%) across all functions.

**Median:** This transformation calculates the median value, which corresponds to the 50th percentile.

| Transformation Name | New formula |
|---|---|
| **Parameter: Formula type** | `Single row formula` |
| **Parameter: Formula** | `median(heightIn)` |
| **Parameter: New column name** | `'medianIn'` |

| Transformation Name | New formula |
|---|---|
| **Parameter: Formula type** | `Single row formula` |
| **Parameter: Formula** | `approximatemedian(heightIn, 0.4)` |
| **Parameter: New column name** | `'approxMedianIn'` |

**Percentile:** This transformation calculates the 68th percentile.

| Transformation Name | New formula |
|---|---|
| **Parameter: Formula type** | `Single row formula` |
| **Parameter: Formula** | `percentile(heightIn, 68, linear)` |
| **Parameter: New column name** | `'percentile68In'` |

| Transformation Name | New formula |
|---|---|
| **Parameter: Formula type** | `Single row formula` |
| **Parameter: Formula** | `approximatepercentile(heightIn, 68, 0.4)` |
| **Parameter: New column name** | `'approxPercentile68In'` |

**Quartile:** This transformation calculates the first quartile, which corresponds to the 25th percentile.

| Transformation Name | New formula |
|---|---|
| **Parameter: Formula type** | `Single row formula` |
| **Parameter: Formula** | `quartile(heightIn, 1, linear)` |
| **Parameter: New column** | `'percentile25In'` |

| | name | |
|---|---|---|
| | | |

| | |
|---|---|
| Transformation Name | New formula |
| Parameter: Formula type | Single row formula |
| Parameter: Formula | approximatequartile(heightIn, 1, 0.4) |
| Parameter: New column name | 'approxPercentile25In' |

**Results:**

| studentId | heightIn | approxPercentile25In | percentile25In | approxPercentile68In | percentile68In | approxMedianIn | |
|---|---|---|---|---|---|---|---|
| 1 | 64 | 64 | 64 | 67.1 | 66.92 | 66 | 6 |
| 2 | 65 | 64 | 64 | 67.1 | 66.92 | 66 | 6 |
| 3 | 63 | 64 | 64 | 67.1 | 66.92 | 66 | 6 |
| 4 | 64 | 64 | 64 | 67.1 | 66.92 | 66 | 6 |
| 5 | 62 | 64 | 64 | 67.1 | 66.92 | 66 | 6 |
| 6 | 66 | 64 | 64 | 67.1 | 66.92 | 66 | 6 |
| 7 | 66 | 64 | 64 | 67.1 | 66.92 | 66 | 6 |
| 8 | 65 | 64 | 64 | 67.1 | 66.92 | 66 | 6 |
| 9 | 69 | 64 | 64 | 67.1 | 66.92 | 66 | 6 |
| 10 | 66 | 64 | 64 | 67.1 | 66.92 | 66 | 6 |
| 11 | 73 | 64 | 64 | 67.1 | 66.92 | 66 | 6 |
| 12 | 69 | 64 | 64 | 67.1 | 66.92 | 66 | 6 |
| 13 | 69 | 64 | 64 | 67.1 | 66.92 | 66 | 6 |
| 14 | 61 | 64 | 64 | 67.1 | 66.92 | 66 | 6 |
| 15 | 64 | 64 | 64 | 67.1 | 66.92 | 66 | 6 |
| 16 | 61 | 64 | 64 | 67.1 | 66.92 | 66 | 6 |
| 17 | 71 | 64 | 64 | 67.1 | 66.92 | 66 | 6 |
| 18 | 67 | 64 | 64 | 67.1 | 66.92 | 66 | 6 |
| 19 | 73 | 64 | 64 | 67.1 | 66.92 | 66 | 6 |
| 20 | 66 | 64 | 64 | 67.1 | 66.92 | 66 | 6 |

# EXAMPLE - POW and SQRT Functions

In this example, you learn how to compute exponentials and square roots on your numeric data.

**Functions**:

| Item | Description |
|------|-------------|
| POW Function | Computes the value of the first argument raised to the value of the second argument. |
| SQRT Function | Computes the square root of the input parameter. Input value can be a Decimal or Integer literal or a reference to a column containing numeric values. All generated values are non-negative. |

**Source:**

The dataset below contains values for x and y:

| X | Y |
|----|----|
| 3 | 4 |
| 4 | 9 |
| 8 | 10 |
| 30 | 40 |

**Transformation:**

You can use the following transformation to generate values for $z^2$.

> **NOTE:** Do not add this step to your recipe right now.

| Transformation Name | New formula |
|---------------------|-------------|
| **Parameter: Formula type** | Single row formula |
| **Parameter: Formula** | (POW(x,2) + POW(y,2)) |
| **Parameter: New column name** | 'Z' |

You can see how column Z is generated as the sum of squares of the other two columns, which yields $z^2$.

Now, edit the transformation to wrap the value computation in a SQRT function. This step is done to compute the value for *z*, which is the distance between the two points based on the Pythagorean theorem.

| Transformation Name | New formula |
|---------------------|-------------|
| **Parameter: Formula type** | Single row formula |
| **Parameter: Formula** | SQRT((POW(x,2) + POW(y,2))) |
| **Parameter: New column name** | 'Z' |

**Results:**

| X | Y | Z |
|---|---|---|
| 3 | 4 | 5 |
| 4 | 9 | 9.848857801796104 |
| 8 | 10 | 12.806248474865697 |
| 30 | 40 | 50 |

# EXAMPLE - PREV Function

This example describes how you can use the PREV function to analyze data that is available in a window in rows before the current one.

**Functions:**

| Item | Description |
|------|-------------|
| PREV Function | Extracts the value from a column that is a specified number of rows before the current value. |
| IF Function | The `IF` function allows you to build if/then/else conditional logic within your transforms. |

The following dataset contains orders for multiple customers over a period of a few days, listed in no particular order. You want to assess how order size has changed for each customer over time and to provide offers to your customers based on changes in order volume.

**Source:**

| Date | CustId | OrderId | OrderValue |
|------|--------|---------|------------|
| 1/4/16 | C001 | Ord002 | 500 |
| 1/11/16 | C003 | Ord005 | 200 |
| 1/20/16 | C002 | Ord007 | 300 |
| 1/21/16 | C003 | Ord008 | 400 |
| 1/4/16 | C001 | Ord001 | 100 |
| 1/7/16 | C002 | Ord003 | 600 |
| 1/8/16 | C003 | Ord004 | 700 |
| 1/21/16 | C002 | Ord009 | 200 |
| 1/15/16 | C001 | Ord006 | 900 |

**Transformation:**

When the data is loaded into the Transformer page, you can use the `PREV` function to gather the order values for the previous two orders into a new column. The trick is to order the `window` transform by the date and group it by customer:

| Transformation Name | Window |
|---------------------|--------|
| Parameter: Formulas | PREV(OrderValue, 1) |
| Parameter: Group by | CustId |
| Parameter: Order by | Date |

| Transformation Name | Window |
|---------------------|--------|
| Parameter: Formulas | PREV(OrderValue, 2) |
| Parameter: Group by | CustId |
| Parameter: Order by | Date |

| Transformation Name | Rename columns |
|---------------------|----------------|

| Parameter: Option | `Manual rename` |
|---|---|
| Parameter: Column | `window` |
| Parameter: New column name | `'OrderValue_1'` |

| Transformation Name | `Rename columns` |
|---|---|
| Parameter: Option | `Manual rename` |
| Parameter: Column | `window1` |
| Parameter: New column name | `'OrderValue_2'` |

You should now have the following columns in your dataset: `Date, CustId, OrderId, OrderValue, OrderValue_1, OrderValue_2`.

The two new columns represent the previous order and the order before that, respectively. Now, each row contains the current order (`OrderValue`) as well as the previous orders. Now, you want to take the following customer actions:

- If the current order is more than 20% greater than the sum of the two previous orders, send a rebate.
- If the current order is less than 90% of the sum of the two previous orders, send a coupon.
- Otherwise, send a holiday card.

To address the first one, you might add the following, which uses the `IF` function to test the value of the current order compared to the previous ones:

| Transformation Name | `New formula` |
|---|---|
| Parameter: Formula type | `Single row formula` |
| Parameter: Formula | `IF(OrderValue >= (1.2 * (OrderValue_1 + OrderValue_2)), 'send rebate', 'no action')` |
| Parameter: New column name | `'CustomerAction'` |

You can now see which customers are due a rebate. Now, edit the above and replace it with the following, which addresses the second condition. If neither condition is valid, then the result is `send holiday card`.

| Transformation Name | `New formula` |
|---|---|
| Parameter: Formula type | `Single row formula` |
| Parameter: Formula | `IF(OrderValue >= (1.2 * (OrderValue_1 + OrderValue_2)), 'send rebate', IF(OrderValue <= (1.2 * (OrderValue_1 + OrderValue_2)), 'send coupon', 'send holiday card'))` |
| Parameter: New column name | `'CustomerAction'` |

**Results:**

After you delete the `OrderValue_1` and `OrderValue_2` columns, your dataset should look like the following. Since the transformations with `PREV` functions grouped by `CustId`, the order of records has changed.

| Date | CustId | OrderId | OrderValue | CustomerAction |
|------|--------|---------|------------|----------------|
| 1/4/16 | C001 | Ord001 | 100 | send rebate |
| 1/7/16 | C001 | Ord002 | 500 | send rebate |
| 1/15/16 | C001 | Ord006 | 900 | send rebate |
| 1/8/16 | C003 | Ord004 | 700 | send rebate |
| 1/11/16 | C003 | Ord005 | 200 | send rebate |
| 1/21/16 | C003 | Ord008 | 400 | send coupon |
| 1/7/16 | C002 | Ord003 | 600 | send rebate |
| 1/20/16 | C002 | Ord007 | 300 | send rebate |
| 1/21/16 | C002 | Ord009 | 200 | send coupon |

# EXAMPLE - Quote Parameter

This example demonstrates how to use `quote` parameter for more sophisticated splitting of columns of data using the `split` transform.

**Source:**

In this example, the following CSV data, which contains contact information, is imported into the application:

```
LastName,FirstName,Role,Company,Address,Status
Wagner,Melody,VP of Engineering,Example.com,"123 Main Street, Oakland, CA 94601",Prospect
Gruber,Hans,"Director, IT",Example.com,"456 Broadway, Burlingame, CA, 94401",Customer
Franks,Mandy,"Sr. Manager, Analytics",Tricorp,"789 Market Street, San Francisco, CA, 94105",Customer
```

**Transformationn:**

When this data is pulled into the application, some initial parsing is performed for you:

| column2 | column3 | column4 | column5 | column6 | column7 |
|---------|---------|---------|---------|---------|---------|
| LastName | FirstName | Role | Company | Address | Status |
| Wagner | Melody | VP of Engineering | Example.com | "123 Main Street, Oakland, CA 94601" | Prospect |
| Gruber | Hans | "Director, IT" | Example.com | "456 Broadway, Burlingame, CA, 94401" | Customer |
| Franks | Mandy | "Sr. Manager, Analytics" | Tricorp | "789 Market Street, San Francisco, CA, 94105" | Customer |

When you open the Recipe Panel, you should see the following transforms:

| | |
|---|---|
| **Transformation Name** | `Split into rows` |
| **Parameter: Column** | `column1` |
| **Parameter: Split on** | `\n` |
| **Parameter: Ignore matches between** | `\"` |
| **Parameter: Quote escape character** | `\"` |

| | |
|---|---|
| **Transformation Name** | `Split column` |
| **Parameter: Column** | `column1` |
| **Parameter: Option** | `On pattern` |
| **Parameter: Match pattern** | `','` |
| **Parameter: Number of Matches** | 5 |
| **Parameter: Ignore matches between** | `\"` |

The first transform splits the raw source data into separate rows in the carriage return character ($\r$), ignoring all values between the double-quote characters. Note that this value must be escaped. The double-quote character does not require escaping. While there are no carriage returns within the actual data, the application recognizes that these double-quotes are identifying single values and adds the quote value.

The second transform splits each row of data into separate columns. Since it is comma-separated data, the application recognizes that this value is the column delimiter, so the `on` value is set to the comma character (`,`). In this case, the quoting is necessary, as there are commas in the values in `column4` and `column6`, which are easy to clean up.

To finish clean up of the dataset, you can promote the first row to be your column headers:

| | |
|---|---|
| **Transformation Name** | `Rename column with row(s)` |
| **Parameter: Option** | `Use row(s) as column names` |
| **Parameter: Type** | `Use a single row to name columns` |
| **Parameter: Row number** | `1` |

You can remove the quotes now. Note that the following applies to two columns:

| | |
|---|---|
| **Transformation Name** | `Replace text or patterns` |
| **Parameter: Column** | `Address,Role` |
| **Parameter: Find** | `'\"'` |
| **Parameter: Replace** | `''` |
| **Parameter: Match all occurrences** | `true` |

Now, you can split up the `Address` column. You can highlight one of the commas and the space after it in the column, but make sure that your final statement looks like the following:

| | |
|---|---|
| **Transformation Name** | `Split column` |
| **Parameter: Column** | `column1` |
| **Parameter: Option** | `On pattern` |
| **Parameter: Match pattern** | `','` |
| **Parameter: Number of Matches** | `2` |

Notice that there is some dirtiness to the resulting `Address3` column:

| Address3 |
|---|
| CA 94601 |
| CA, 94401 |
| CA, 94105 |

Use the following to remove the comma. In this case, it's important to leave the space between the two values in the column, so the `on` value should only be a comma. Below, the `width` value is two single quotes:

| | |
|---|---|
| **Transformation Name** | `Replace text or patterns` |
| **Parameter: Column** | `Address3` |
| **Parameter: Find** | `','` |

| Parameter: Replace | '' |
|---|---|
| Parameter: Match all occurrences | `true` |

You can now split the `Address3` column on the space delimiter:

| Transformation Name | `Split by delimiter` |
|---|---|
| Parameter: Column | `Address3` |
| Parameter: Option | `by delimiter` |
| Parameter: Delimiter | `' '` |
| Parameter: Number of columns to create | 2 |

**Results:**

After you rename the columns, you should see the following:

| LastName | FirstName | Role | Company | Address | City | State | Zipcode | Status |
|---|---|---|---|---|---|---|---|---|
| Wagner | Melody | VP of Engineering | Example.com | 123 Main Street | Oakland | CA | 94601 | Prospect |
| Gruber | Hans | Director, IT | Example.com | 456 Broadway | Burlingame | CA | 94401 | Customer |
| Franks | Mandy | Sr. Manager, Analytics | Tricorp | 789 Market Street | San Francisco | CA | 94105 | Customer |

# EXAMPLE - RANDBETWEEN and PI Functions

This example illustrates how you can apply functions to generate random numeric data in your dataset.

**Functions**:

| Item | Description |
|------|-------------|
| RANDBETWEEN Function | Generates a random integer between a low and a high number. Two inputs may be Integer or Decimal types, functions returning these types, or column references. |
| PI Function | The `PI` function generates the value of pi to 15 decimal places: 3.1415926535897932. |
| ROUND Function | Rounds input value to the nearest integer. Input can be an Integer, a Decimal, a column reference, or an expression. Optional second argument can be used to specify the number of digits to which to round. |
| TRUNC Function | Removes all digits to the right of the decimal point for any value. Optionally, you can specify the number of digits to which to round. Input can be an Integer, a Decimal, a column reference, or an expression. |
| POW Function | Computes the value of the first argument raised to the value of the second argument. |

## Source:

In the following example, a company produces 10 circular parts, the size of which is measured in each product's radius in inches.

| prodId | radius_in |
|--------|-----------|
| p001 | 1 |
| p002 | 2 |
| p003 | 3 |
| p004 | 4 |
| p005 | 5 |
| p006 | 6 |
| p007 | 7 |
| p008 | 8 |
| p009 | 9 |
| p010 | 10 |

Based on the above data, the company wants to generate some additional sizing information for these circular parts, including the generation of two points along each part's circumference where quality stress tests can be applied.

## Transformation:

To begin, you can use the following steps to generate the area and circumference for each product, rounded to three decimal points:

| Transformation Name | New formula |
|---------------------|-------------|
| **Parameter: Formula type** | `Single row formula` |
| **Parameter: Formula** | `ROUND(PI() * (POW(radius_in, 2)), 3)` |
| **Parameter: New column name** | `'area_sqin'` |

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | ROUND(PI() * (2 * radius_in), 3) |
| Parameter: New column name | 'circumference_in' |

For quality purposes, the company needs two tests points along the circumference, which are generated by calculating two separate random locations along the circumference. Since the RANDBETWEEN function only calculates using Integer values, you must first truncate the values from circumference_in:

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | TRUNC(circumference_in) |
| Parameter: New column name | 'trunc_circumference_in' |

Then, you can calculate the random points using the following:

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | RANDBETWEEN(0, trunc_circumference_in) |
| Parameter: New column name | 'testPt01_in' |

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | RANDBETWEEN(0, trunc_circumference_in) |
| Parameter: New column name | 'testPt02_in' |

**Results:**

After the trunc_circumference_in column is dropped, the data should look similar to the following:

| prodId | radius_in | area_sq_in | circumference_in | testPt01_in | testPt02_in |
|---|---|---|---|---|---|
| p001 | 1 | 3.142 | 6.283 | 5 | 5 |
| p002 | 2 | 12.566 | 12.566 | 3 | 3 |
| p003 | 3 | 28.274 | 18.850 | 13 | 13 |
| p004 | 4 | 50.265 | 25.133 | 24 | 24 |
| p005 | 5 | 78.540 | 31.416 | 0 | 0 |
| p006 | 6 | 113.097 | 37.699 | 15 | 15 |
| p007 | 7 | 153.938 | 43.982 | 11 | 11 |
| p008 | 8 | 201.062 | 50.265 | 1 | 1 |

| p009 | 9 | 254.469 | 56.549 | 29 | 29 |
| p010 | 10 | 314.159 | 62.832 | 21 | 21 |

# EXAMPLE - RANK Functions

This example demonstrates you to generate a ranked order of values.

**Functions**:

| Item | Description |
|------|-------------|
| RANK Function | Computes the rank of an ordered set of value within groups. Tie values are assigned the same rank, and the next ranking is incremented by the number of tie values. |
| DENSERANK Function | Computes the rank of an ordered set of value within groups. Tie values are assigned the same rank, and the next ranking is incremented by 1. |

**Source:**

The following dataset contains lap times for three racers in a four-lap race. Note that for some racers, there are tie values for lap times.

| Runner | Lap | Time |
|--------|-----|------|
| Dave | 1 | 72.2 |
| Dave | 2 | 73.31 |
| Dave | 3 | 72.2 |
| Dave | 4 | 70.85 |
| Mark | 1 | 71.73 |
| Mark | 2 | 71.73 |
| Mark | 3 | 72.99 |
| Mark | 4 | 70.63 |
| Tom | 1 | 74.43 |
| Tom | 2 | 70.71 |
| Tom | 3 | 71.02 |
| Tom | 4 | 72.98 |

**Transformation:**

You can apply the `RANK()` function to the `Time` column, grouped by individual runner:

| Transformation Name | Window |
|---------------------|--------|
| Parameter: Formulas | RANK() |
| Parameter: Group by | Runner |
| Parameter: Order by | Time |

You can use the `DENSERANK()` function on the same column, grouping by runner:

| Transformation Name | Window |
|---------------------|--------|
| Parameter: Formulas | DENSERANK() |

| Parameter: Group by | Runner |
|---|---|
| Parameter: Order by | Time |

**Results:**

After renaming the columns, you have the following output:

| Runner | Lap | Time | Rank | Rank-Dense |
|---|---|---|---|---|
| Mark | 4 | 70.63 | 1 | 1 |
| Mark | 1 | 71.73 | 2 | 2 |
| Mark | 2 | 71.73 | 2 | 2 |
| Mark | 3 | 72.99 | 4 | 3 |
| Tom | 2 | 70.71 | 1 | 1 |
| Tom | 3 | 71.02 | 2 | 2 |
| Tom | 4 | 72.98 | 3 | 3 |
| Tom | 1 | 74.43 | 4 | 4 |
| Dave | 4 | 70.85 | 1 | 1 |
| Dave | 1 | 72.2 | 2 | 2 |
| Dave | 3 | 72.2 | 2 | 2 |
| Dave | 2 | 73.31 | 4 | 3 |

# EXAMPLE - Replacement Transforms

This example illustrates the different uses of the replacement transformations to replace or extract cell data.

**Source:**

The following dataset contains contact information that has been gathered by your marketing platform from actions taken by visitors on your website. You must clean up this data and prepare it for use in an analytics platform.

| LeadId | LastName | FirstName | Title | Phone | Request |
|---|---|---|---|---|---|
| LE160301001 | Jones | Charles | Chief Technical Officer | 415-555-1212 | reg |
| LE160301002 | Lyons | Edward | | 415-012-3456 | download whitepaper |
| LE160301003 | Martin | Mary | CEO | 510-555-5555 | delete account |
| LE160301004 | Smith | Talia | Engineer | 510-123-4567 | free trial |

**Transformation:**

**Title column:** For example, you first notice that some data is missing. Your analytics platform recognizes the string value, "`#MISSING#`" as an indicator of a missing value. So, you click the missing values bar in the Title column. Then, you select the Replace suggestion card. Note that the default replacement is a null value, so you click **Edit** and update it:

| | |
|---|---|
| **Transformation Name** | Edit column with formula |
| **Parameter: Columns** | Title |
| **Parameter: Formula** | if(ismissing([Title]),'#MISSING#',Title) |

**Request column:** In the Request column, you notice that the `reg` entry should be cleaned up. Add the following transformation, which replaces that value:

| | |
|---|---|
| **Transformation Name** | Replace text or pattern |
| **Parameter: Column** | Request |
| **Parameter: Find** | `` `{start}reg{end}` `` |
| **Parameter: Replace with** | Registration |

The above transformation uses a Pattern as the expression of the `on:` parameter. This expression indicates to match from the start of the cell value, the string literal `reg`, and then the end of the cell value, which matches on complete cell values of `reg` only.

This transformation works great on the sample, but what happens if the value is `Reg` with a capital `R`? That value might not be replaced. To improve the transformation, you can modify the transformation with the following Pattern in the `on` parameter, which captures differences in capitalization:

| | |
|---|---|
| **Transformation Name** | Replace text or pattern |
| **Parameter: Column** | Request |
| **Parameter: Find** | `` `{start}{[R|r]}eg{end}` `` |
| | |

| | |
|---|---|
| **Parameter: Replace with** | `'Registration'` |

Add the above transformation to your recipe. Then, it occurs to you that all of the values in the `Request` column should be capitalized in title or proper case:

| | |
|---|---|
| **Transformation Name** | `Edit column with formula` |
| **Parameter: Columns** | `Request` |
| **Parameter: Formula** | `proper(Request)` |

Now, all values are capitalized as titles.

**Phone column:** You might have noticed some issues with the values in the `Phone` column. In the United States, the prefix `555` is only used for gathering information; these are invalid phone numbers.

In the data grid, you select the first instance of `555` in the column. However, it selects all instances of that pattern, including ones that you don't want to modify. In this case, continue your selection by selecting the similar instance of `555` in the other row. In the suggestion cards, you click the Replace Text or Pattern transformation.

Notice, however, that the default Replace Text or Pattern transformation has also highlighted the second `555` pattern in one instance, which could be a problem in other phone numbers not displayed in the sample. You must modify the selection pattern for this transformation. In the `on:` parameter below, the Pattern has been modified to match only the instances of `555` that appear in the second segment in the phone number format:

| | |
|---|---|
| **Transformation Name** | `Replace text or pattern` |
| **Parameter: Column** | `Phone` |
| **Parameter: Find** | `` `{start}%{3}-555-%*{end}` `` |
| **Parameter: Replace with** | `'#INVALID#'` |
| **Parameter: Match all occurrences** | `true` |

Note the wildcard construct has been added (`%*`). While it might be possible to add a pattern that matches on the last four characters exactly (`%{4}`), that matching pattern would not capture the possibility of a phone number having an extension at the end of it. The above expression does.

> **NOTE:** The above transformation creates values that are mismatched with the Phone Number data type. In this example, however, these mismatches are understood to be for the benefit of the system consuming your Trifacta output.

**LeadId column:** You might have noticed that the lead identifier column (`LeadId`) contains some embedded information: a date value and an identifier for the instance within the day. The following steps can be used to break out this information. The first one creates a separate working column with this information, which allows us to preserve the original, unmodified column:

| | |
|---|---|
| **Transformation Name** | `New formula` |
| **Parameter: Formula type** | `Single row formula` |
| **Parameter: Formula** | `LeadId` |
| **Parameter: New column name** | `'LeadIdworking'` |

You can now work off of this column to create your new ones. First, you can use the following replace transformation to remove the leading two characters, which are not required for the new columns:

| Transformation Name | Replace text or pattern |
|---|---|
| Parameter: Column | LeadIdworking |
| Parameter: Find | 'LE' |
| Parameter: Replace with | '' |

Notice that the date information is now neatly contained in the first characters of the working column. Use the following to extract these values to a new column:

| Transformation Name | Extract text or pattern |
|---|---|
| Parameter: Column to extract from | LeadIdworking |
| Parameter: Option | Custom text or pattern |
| Parameter: Text to extract | `{start}%{6}` |

The new `LeadIdworking2` column now contains only the date information. Cleaning up this column requires reformatting the data, retyping it as a Datetime type, and then applying the `dateformat` function to format it to your satisfaction. These steps are left as a separate exercise.

For now, let's just rename the column:

| Transformation Name | Rename columns |
|---|---|
| Parameter: Option | Manual rename |
| Parameter: Column | LeadIdworking1 |
| Parameter: New column name | 'LeadIdDate' |

In the first working column, you can now remove the date information using the following:

| Transformation Name | Replace text or pattern |
|---|---|
| Parameter: Column | LeadIdworking |
| Parameter: Find | `{start}%{6}` |
| Parameter: Replace with | '' |

You can rename this column to indicate it is a daily identifier:

| Transformation Name | Rename columns |
|---|---|
| Parameter: Option | Manual rename |
| Parameter: Column | LeadIdworking |
| Parameter: New column name | 'LeadIdDaily' |

**Results:**

| LeadId | LeadIdDaily | LeadIdDate | LastName | FirstName | Title | Phone | Request |
|---|---|---|---|---|---|---|---|
| LE160301001 | 001 | 160301 | Jones | Charles | Chief Technical Officer | #INVALID# | Registration |
| LE160301002 | 002 | 160301 | Lyons | Edward | #MISSING# | 415-012-3456 | Download Whitepaper |
| LE160301003 | 003 | 160301 | Martin | Mary | CEO | #INVALID# | Delete Account |
| LE160301004 | 004 | 160301 | Smith | Talia | Engineer | 510-123-4567 | Free Trial |

# EXAMPLE - Rolling Date Functions

This example describes how to use rolling functions for Datetime values.

**Functions**:

| Item | Description |
| --- | --- |
| ROLLINGMINDATE Function | Computes the rolling minimum of Date values forward or backward of the current row within the specified column. Inputs must be of Datetime type. |
| ROLLINGMAXDATE Function | Computes the rolling maximum of date values forward or backward of the current row within the specified column. Inputs must be of Datetime type. |
| ROLLINGMODEDATE Function | Computes the rolling mode (most common value) forward or backward of the current row within the specified column. Input values must be of Datetime data type. |

**Source:**

The following table contains an unordered list of orders:

| myDate | prodId | orderDollars |
| --- | --- | --- |
| 2020-03-13 | p001 | 1445 |
| 2020-03-06 | p002 | 712 |
| 2020-03-16 | p003 | 1374 |
| 2020-03-23 | p001 | 1675 |
| 2020-04-09 | p002 | 1005 |
| 2020-08-09 | p003 | 984 |
| 2020-05-02 | p001 | 1395 |
| 2020-06-14 | p002 | 1866 |
| 2020-07-16 | p003 | 824 |
| 2020-09-02 | p001 | 1785 |
| 2020-08-31 | p002 | 697 |
| 2020-10-22 | p003 | 1513 |
| 2020-03-17 | p001 | 768 |
| 2020-03-21 | p002 | 1893 |
| 2020-03-23 | p003 | 1122 |
| 2020-04-06 | p001 | 805 |
| 2020-05-09 | p002 | 1752 |
| 2021-01-09 | p003 | 616 |
| 2020-08-18 | p001 | 1563 |
| 2020-09-12 | p002 | 730 |
| 2020-10-04 | p003 | 587 |
| 2021-02-15 | p001 | 1979 |
| 2021-02-22 | p002 | 134 |
| 2021-03-14 | p003 | 938 |

**Transformation:**

You can use the following Window transformation to calculate the rolling minimum, maximum, and mode dates for the last five orders for each product identifier:

| Transformation Name | `Window` |
|---|---|
| Parameter: Formula1 | `ROLLINGMINDATE(orderDate, 4, 0)` |
| Parameter: Formula2 | `ROLLINGMAXDATE(orderDate, 4, 0)` |
| Parameter: Formula3 | `ROLLINGMODEDATE(orderDate, 4, 0)` |
| Parameter: Group by | `prodId` |
| Parameter: Order by | `prodId` |

You can use the following transformation to rename the generated `window` columns:

| Transformation Name | `Rename columns` |
|---|---|
| Parameter: Option | `Manual rename` |
| Parameter: Column | `window1` |
| Parameter: New column name | `rollingMinDate` |
| Parameter: Parameter: Column | `window2` |
| Parameter: New column name | `rollingMaxDate` |
| Parameter: Parameter: Column | `window3` |
| Parameter: New column name | `rollingModeDate` |

**Results:**

| orderDate | prodId | orderDollars | rollingMinDate | rollingMaxDate | rollingModeDate |
|---|---|---|---|---|---|
| 3/16/20 | p003 | 1374 | 3/16/20 | 3/16/20 | 3/16/20 |
| 8/9/20 | p003 | 984 | 3/16/20 | 8/9/20 | 3/16/20 |
| 7/16/20 | p003 | 824 | 3/16/20 | 8/9/20 | 3/16/20 |
| 10/22/20 | p003 | 1513 | 3/16/20 | 10/22/20 | 3/16/20 |
| 3/23/20 | p003 | 1122 | 3/16/20 | 10/22/20 | 3/16/20 |
| 1/9/21 | p003 | 616 | 3/23/20 | 1/9/21 | 3/23/20 |
| 10/4/20 | p003 | 587 | 3/23/20 | 1/9/21 | 3/23/20 |
| 3/14/21 | p003 | 938 | 3/23/20 | 3/14/21 | 3/23/20 |
| 3/13/20 | p001 | 1445 | 3/13/20 | 3/13/20 | 3/13/20 |
| 3/23/20 | p001 | 1675 | 3/13/20 | 3/23/20 | 3/13/20 |
| 5/2/20 | p001 | 1395 | 3/13/20 | 5/2/20 | 3/13/20 |
| 9/2/20 | p001 | 1785 | 3/13/20 | 9/2/20 | 3/13/20 |
| 3/17/20 | p001 | 768 | 3/13/20 | 9/2/20 | 3/13/20 |

| | | | | | |
|---|---|---|---|---|---|
| 4/6/20 | p001 | 805 | 3/17/20 | 9/2/20 | 3/17/20 |
| 8/18/20 | p001 | 1563 | 3/17/20 | 9/2/20 | 3/17/20 |
| 2/15/21 | p001 | 1979 | 3/17/20 | 2/15/21 | 3/17/20 |
| 3/6/20 | p002 | 712 | 3/6/20 | 3/6/20 | 3/6/20 |
| 4/9/20 | p002 | 1005 | 3/6/20 | 4/9/20 | 3/6/20 |
| 6/14/20 | p002 | 1866 | 3/6/20 | 6/14/20 | 3/6/20 |
| 8/31/20 | p002 | 697 | 3/6/20 | 8/31/20 | 3/6/20 |
| 3/21/20 | p002 | 1893 | 3/6/20 | 8/31/20 | 3/6/20 |
| 5/9/20 | p002 | 1752 | 3/21/20 | 8/31/20 | 3/21/20 |
| 9/12/20 | p002 | 730 | 3/21/20 | 9/12/20 | 3/21/20 |
| 2/22/21 | p002 | 134 | 3/21/20 | 2/22/21 | 3/21/20 |

# EXAMPLE - Rolling Functions

This example describes how to use rolling computational functions.

**Functions**:

| Item | Description |
|------|-------------|
| ROLLINGSUM Function | Computes the rolling sum of values forward or backward of the current row within the specified column. |
| ROLLINGAVERAGE Function | Computes the rolling average of values forward or backward of the current row within the specified column. |
| ROWNUMBER Function | Generates a new column containing the row number as sorted by the `order` parameter and optionally grouped by the `group` parameter. |

Also:

| Item | Description |
|------|-------------|
| MONTH Function | Derives the month integer value from a Datetime value. Source value can be a a reference to a column containing Datetime values or a literal. |
| FLOOR Function | Computes the largest integer that is not more than the input value. Input can be an Integer, a Decimal, a column reference, or an expression. |
| ROWNUMBER Function | Generates a new column containing the row number as sorted by the `order` parameter and optionally grouped by the `group` parameter. |

The following dataset contains sales data over the final quarter of the year.

**Source:**

| Date | Sales |
|------|-------|
| 10/2/16 | 200 |
| 10/9/16 | 500 |
| 10/16/16 | 350 |
| 10/23/16 | 400 |
| 10/30/16 | 190 |
| 11/6/16 | 550 |
| 11/13/16 | 610 |
| 11/20/16 | 480 |
| 11/27/16 | 660 |
| 12/4/16 | 690 |
| 12/11/16 | 810 |
| 12/18/16 | 950 |
| 12/25/16 | 1020 |
| 1/1/17 | 680 |

**Transformation:**

First, you want to maintain the row information as a separate column. Since data is ordered already by the `Date` column, you can use the following:

| Transformation Name | Window |
|---|---|
| Parameter: Formulas | ROWNUMBER() |
| Parameter: Order by | Date |

Rename this column to `rowId` for week of quarter.

Now, you want to extract month and week information from the `Date` values. Deriving the month value:

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | MONTH(Date) |
| Parameter: New column name | 'Month' |

Deriving the quarter value:

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | (1 + FLOOR(((month-1)/3))) |
| Parameter: New column name | 'QTR' |

Deriving the week-of-quarter value:

| Transformation Name | Window |
|---|---|
| Parameter: Formulas | ROWNUMBER() |
| Parameter: Group by | QTR |
| Parameter: Order by | Date |

Rename this column `WOQ` (week of quarter).

Deriving the week-of-month value:

| Transformation Name | Window |
|---|---|
| Parameter: Formulas | ROWNUMBER() |
| Parameter: Group by | Month |
| Parameter: Order by | Date |

Rename this column `WOM` (week of month).

Now, you perform your rolling computations. Compute the running total of sales using the following:

| Transformation Name | Window |
|---|---|
| Parameter: Formulas | ROLLINGSUM(Sales, -1, 0) |
| Parameter: Group by | QTR |
| Parameter: Order by | Date |

The -1 parameter is used in the above computation to gather the rolling sum of all rows of data from the current one to the first one. Note that the use of the QTR column for grouping, which moves the value for the 01/01/2017 into its own computational bucket. This may or may not be preferred.

Rename this column QTD (quarter to-date). Now, generate a similar column to compute the rolling average of weekly sales for the quarter:

| Transformation Name | Window |
|---|---|
| Parameter: Formulas | ROUND(ROLLINGAVERAGE(Sales, -1, 0)) |
| Parameter: Group by | QTR |
| Parameter: Order by | Date |

Since the ROLLINGAVERAGE function can compute fractional values, it is wrapped in the ROUND function for neatness. Rename this column avgWeekByQuarter.

**Results:**

When the unnecessary columns are dropped and some reordering is applied, your dataset should look like the following:

| Date | WOQ | Sales | QTD | avgWeekByQuarter |
|---|---|---|---|---|
| 10/2/16 | 1 | 200 | 200 | 200 |
| 10/9/16 | 2 | 500 | 700 | 350 |
| 10/16/16 | 3 | 350 | 1050 | 350 |
| 10/23/16 | 4 | 400 | 1450 | 363 |
| 10/30/16 | 5 | 190 | 1640 | 328 |
| 11/6/16 | 6 | 550 | 2190 | 365 |
| 11/13/16 | 7 | 610 | 2800 | 400 |
| 11/20/16 | 8 | 480 | 3280 | 410 |
| 11/27/16 | 9 | 660 | 3940 | 438 |
| 12/4/16 | 10 | 690 | 4630 | 463 |
| 12/11/16 | 11 | 810 | 5440 | 495 |
| 12/18/16 | 12 | 950 | 6390 | 533 |
| 12/25/16 | 13 | 1020 | 7410 | 570 |
| 1/1/17 | 1 | 680 | 680 | 680 |

# EXAMPLE - Rolling Functions 2

This example describes how to use rolling statistical functions.

**Functions**:

| Item | Description |
|------|-------------|
| ROLLINGAVERAGE Function | Computes the rolling average of values forward or backward of the current row within the specified column. |
| ROLLINGMAX Function | Computes the rolling maximum of values forward or backward of the current row within the specified column. Input s can be Integer, Decimal, or Datetime. |
| ROLLINGSTDEV Function | Computes the rolling standard deviation of values forward or backward of the current row within the specified column. |
| ROLLINGVAR Function | Computes the rolling variance of values forward or backward of the current row within the specified column. |
| ROLLINGSTDEVSAMP Function | Computes the rolling standard deviation of values forward or backward of the current row within the specified column using the sample statistical method. |
| ROLLINGVARSAMP Function | Computes the rolling variance of values forward or backward of the current row within the specified column using the sample statistical method. |

Also:

| Item | Description |
|------|-------------|
| MERGE Function | Merges two or more columns of String type to generate output of String type. Optionally, you can insert a delimiter between the merged values. |
| ROUND Function | Rounds input value to the nearest integer. Input can be an Integer, a Decimal, a column reference, or an expression. Optional second argument can be used to specify the number of digits to which to round. |

**Source:**

In this example, the following data comes from times recorded at regular intervals during a three-lap race around a track. The source data is in cumulative time in seconds (`time_sc`). You can use ROLLING and other windowing functions to break down the data into more meaningful metrics.

| lap | quarter | time_sc |
|-----|---------|---------|
| 1 | 0 | 0.000 |
| 1 | 1 | 19.554 |
| 1 | 2 | 39.785 |
| 1 | 3 | 60.021 |
| 2 | 0 | 80.950 |
| 2 | 1 | 101.785 |
| 2 | 2 | 121.005 |
| 2 | 3 | 141.185 |
| 3 | 0 | 162.008 |
| 3 | 1 | 181.887 |
| 3 | 2 | 200.945 |
| 3 | 3 | 220.856 |

**Transformation:**

**Primary key:** Since the quarter information repeats every lap, there is no unique identifier for each row. The following steps create this identifier:

| Transformation Name | Change column data type |
|---|---|
| Parameter: Columns | lap,quarter |
| Parameter: New type | String |

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | MERGE(['l',lap,'q',quarter]) |
| Parameter: New column name | 'splitId' |

**Get split times:** Use the following transform to break down the splits for each quarter of the race:

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Multiple row formula |
| Parameter: Formula | ROUND(time_sc - PREV(time_sc, 1), 3) |
| Parameter: Order rows by | splitId |
| Parameter: New column name | 'split_time_sc' |

**Compute rolling computations:** You can use the following types of computations to provide rolling metrics on the current and three previous splits:

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Multiple row formula |
| Parameter: Formula | ROLLINGAVERAGE(split_time_sc, 3) |
| Parameter: Order rows by | splitId |
| Parameter: New column name | 'ravg' |

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Multiple row formula |
| Parameter: Formula | ROLLINGMAX(split_time_sc, 3) |
| Parameter: Order rows by | splitId |
| Parameter: New column name | 'rmax' |

| Transformation Name | New formula |
|---|---|

| Parameter: Formula type | Multiple row formula |
|---|---|
| Parameter: Formula | ROLLINGMIN(split_time_sc, 3) |
| Parameter: Order rows by | splitId |
| Parameter: New column name | 'rmin' |

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Multiple row formula |
| Parameter: Formula | ROUND(ROLLINGSTDEV(split_time_sc, 3), 3) |
| Parameter: Order rows by | splitId |
| Parameter: New column name | 'rstdev' |

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Multiple row formula |
| Parameter: Formula | ROUND(ROLLINGVAR(split_time_sc, 3), 3) |
| Parameter: Order rows by | splitId |
| Parameter: New column name | 'rvar' |

**Compute rolling computations using sample method:** These metrics compute the rolling STDEV and VAR on the current and three previous splits using the sample method:

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Multiple row formula |
| Parameter: Formula | ROUND(ROLLINGSTDEVSAMP(split_time_sc, 3), 3) |
| Parameter: Order rows by | splitId |
| Parameter: New column name | 'rstdev_samp' |

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Multiple row formula |
| Parameter: Formula | ROUND(ROLLINGVARSAMP(split_time_sc, 3), 3) |
| Parameter: Order rows by | splitId |
| Parameter: New column name | 'rvar_samp' |

### Results:

When the above transforms have been completed, the results look like the following:

| lap | quarter | splitId | time_sc | split_time_sc | rvar_samp | rstdev_samp | rvar | rstdev | rmin | rmax | ravg |
|---|---|---|---|---|---|---|---|---|---|---|---|

| | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 0 | l1q0 | 0 | | | | | | | | |
| 1 | 1 | l1q1 | 20.096 | 20.096 | | | 0 | 0 | 20.096 | 20.096 | 20.096 |
| 1 | 2 | l1q2 | 40.53 | 20.434 | 0.229 | 0.479 | 0.029 | 0.169 | 20.096 | 20.434 | 20.265 |
| 1 | 3 | l1q3 | 61.031 | 20.501 | 0.154 | 0.392 | 0.031 | 0.177 | 20.096 | 20.501 | 20.344 |
| 2 | 0 | l2q0 | 81.087 | 20.056 | 0.315 | 0.561 | 0.039 | 0.198 | 20.056 | 20.501 | 20.272 |
| 2 | 1 | l2q1 | 101.383 | 20.296 | 0.142 | 0.376 | 0.029 | 0.17 | 20.056 | 20.501 | 20.322 |
| 2 | 2 | l2q2 | 122.092 | 20.709 | 0.617 | 0.786 | 0.059 | 0.242 | 20.056 | 20.709 | 20.39 |
| 2 | 3 | l2q3 | 141.886 | 19.794 | 0.621 | 0.788 | 0.113 | 0.337 | 19.794 | 20.709 | 20.214 |
| 3 | 0 | l3q0 | 162.581 | 20.695 | 0.579 | 0.761 | 0.139 | 0.373 | 19.794 | 20.709 | 20.373 |
| 3 | 1 | l3q1 | 183.018 | 20.437 | 0.443 | 0.666 | 0.138 | 0.371 | 19.794 | 20.709 | 20.409 |
| 3 | 2 | l3q2 | 203.493 | 20.475 | 0.537 | 0.733 | 0.113 | 0.336 | 19.794 | 20.695 | 20.35 |
| 3 | 3 | l3q3 | 222.893 | 19.4 | 0.520 | 0.721 | 0.252 | 0.502 | 19.4 | 20.695 | 20.252 |

You can reduce the number of steps by applying a `window` transform such as the following:

| | |
|---|---|
| **Transformation Name** | `Window` |
| **Parameter: Formula1** | `lap` |
| **Parameter: Formula2** | `rollingaverage(split_time_sc, 0, 3)` |
| **Parameter: Formula3** | `rollingmax(split_time_sc, 0, 3)` |
| **Parameter: Formula4** | `rollingmin(split_time_sc, 0, 3)` |
| **Parameter: Formula5** | `round(rollingstdev(split_time_sc, 0, 3), 3)` |
| **Parameter: Formula6** | `round(rollingvar(split_time_sc, 0, 3), 3)` |
| **Parameter: Formula7** | `round(rollingstdevsamp(split_time_sc, 0, 3), 3)` |
| **Parameter: Formula8** | `round(rollingvarsamp(split_time_sc, 0, 3), 3)` |
| **Parameter: Group by** | `lap` |
| **Parameter: Order by** | `lap` |

However, you must rename all of the generated `windowX` columns.

# EXAMPLE - ROLLINGKTHLARGEST Functions

This example describes how to use rolling kthlargest functions for calculating ranking of values within a defined window of rows.

**Functions**:

| Item | Description |
|------|-------------|
| ROLLINGKTHLARGEST Function | Computes the rolling *kth* largest value forward or backward of the current row. Inputs can be Integer, Decimal, or Datetime. |
| ROLLINGKTHLARGESTUNIQUE Function | Computes the rolling unique *kth* largest value forward or backward of the current row. Inputs can be Integer, Decimal, or Datetime. |
| ROWNUMBER Function | Generates a new column containing the row number as sorted by the `order` parameter and optionally grouped by the `group` parameter. |

The following dataset contains daily counts of server restarts across three servers over the preceding week. High server restart counts can indicate poor server health. In this example, you are interested in knowing for each server the rolling highest and second highest count of restarts per server over the previous week.

**Source:**

| Date | Server | Restarts |
|------|--------|----------|
| 2/21/18 | s01 | 4 |
| 2/21/18 | s02 | 0 |
| 2/21/18 | s03 | 0 |
| 2/22/18 | s01 | 4 |
| 2/22/18 | s02 | 1 |
| 2/22/18 | s03 | 2 |
| 2/23/18 | s01 | 2 |
| 2/23/18 | s02 | 3 |
| 2/23/18 | s03 | 4 |
| 2/24/18 | s01 | 1 |
| 2/24/18 | s02 | 0 |
| 2/24/18 | s03 | 2 |
| 2/25/18 | s01 | 5 |
| 2/25/18 | s02 | 0 |
| 2/25/18 | s03 | 4 |
| 2/26/18 | s01 | 1 |
| 2/26/18 | s02 | 2 |
| 2/26/18 | s03 | 1 |
| 2/27/18 | s01 | 1 |
| 2/27/18 | s02 | 2 |
| 2/27/18 | s03 | 2 |

**Transformation:**

First, you want to maintain the row information as a separate column. Since data is ordered already by the `Date` column, you can use the following:

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | ROWNUMBER() |
| Parameter: New column name | 'entryId' |

Use the following function to compute the rolling *kth* largest value of server restarts per server over the previous week. In this case, you can use the `ROLLINGKTHLARGEST` function, setting *k*=1. Uniqueness doesn't matter for calculating the highest value:

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Multiple row formula |
| Parameter: Formula | rollingkthlargest(Restarts, 1, 6, 0) |
| Parameter: Sort Rows by | Server |
| Parameter: Group Rows by | Server |
| Parameter: New column name | 'rollingkthlargest_1' |

Use the following function to compute the rolling second highest value. In this case, you can use `ROLLINGKTHLARGESTUNIQUE`:

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Multiple row formula |
| Parameter: Formula | rollingkthlargestunique(Restarts, 2, 6, 0) |
| Parameter: Sort Rows by | Server |
| Parameter: Group Rows by | Server |
| Parameter: New column name | 'rollingKthLargestUnique_2' |

**Results:**

| entryId | Date | Server | Restarts | rollingKthLargestUnique_2 | rollingkthlargest_Restarts |
|---|---|---|---|---|---|
| 3 | 2/21/18 | s02 | 0 | 0 | 0 |
| 6 | 2/22/18 | s02 | 1 | 0 | 1 |
| 9 | 2/23/18 | s02 | 3 | 1 | 3 |
| 12 | 2/24/18 | s02 | 0 | 1 | 3 |
| 15 | 2/25/18 | s02 | 0 | 1 | 3 |
|  |  |  |  |  |  |

| 18 | 2/26/18 | s02 | 2 | 2 | 3 |
|---|---|---|---|---|---|
| 21 | 2/27/18 | s02 | 2 | 2 | 3 |
| 4 | 2/21/18 | s03 | 0 | 0 | 0 |
| 7 | 2/22/18 | s03 | 2 | 0 | 2 |
| 10 | 2/23/18 | s03 | 4 | 2 | 4 |
| 13 | 2/24/18 | s03 | 2 | 2 | 4 |
| 16 | 2/25/18 | s03 | 4 | 2 | 4 |
| 19 | 2/26/18 | s03 | 1 | 2 | 4 |
| 22 | 2/27/18 | s03 | 2 | 2 | 4 |
| 2 | 2/21/18 | s01 | 4 | 4 | 4 |
| 5 | 2/22/18 | s01 | 4 | 4 | 4 |
| 8 | 2/23/18 | s01 | 2 | 2 | 4 |
| 11 | 2/24/18 | s01 | 1 | 2 | 4 |
| 14 | 2/25/18 | s01 | 5 | 4 | 5 |
| 17 | 2/26/18 | s01 | 1 | 4 | 5 |
| 20 | 2/27/18 | s01 | 1 | 4 | 5 |

# EXAMPLE - Rounding Functions

This example demonstrates how the rounding functions work together.

**Functions**:

| Item | Description |
|------|-------------|
| FLOOR Function | Computes the largest integer that is not more than the input value. Input can be an Integer, a Decimal, a column reference, or an expression. |
| CEILING Function | Computes the **ceiling** of a value, which is the smallest integer that is greater than the input value. Input can be an Integer, a Decimal, a column reference, or an expression. |
| ROUND Function | Rounds input value to the nearest integer. Input can be an Integer, a Decimal, a column reference, or an expression. Optional second argument can be used to specify the number of digits to which to round. |
| MOD Function | Returns the modulo value, which is the remainder of dividing the first argument by the second argument. Equivalent to the % operator. |

**Source:**

| rowNum | X |
|--------|------|
| 1 | -2.5 |
| 2 | -1.2 |
| 3 | 0 |
| 4 | 1 |
| 5 | 1.5 |
| 6 | 2.5 |
| 7 | 3.9 |
| 8 | 4 |
| 9 | 4.1 |
| 10 | 11 |

**Transformation:**

| Transformation Name | New formula |
|---------------------|-------------|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | FLOOR(X) |
| Parameter: New column name | 'floorX' |

| Transformation Name | New formula |
|---------------------|-------------|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | CEILING(X) |
| Parameter: New column name | 'ceilingX' |

| Transformation Name | New formula |
|---------------------|-------------|

| Parameter: Formula type | Single row formula |
|---|---|
| Parameter: Formula | ROUND (X) |
| Parameter: New column name | 'roundX' |

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | (X % 2) |
| Parameter: New column name | 'modX' |

**Results:**

| rowNum | X | modX | roundX | ceilingX | floorX |
|---|---|---|---|---|---|
| 1 | -2.5 | | -2 | -2 | -3 |
| 2 | -1.2 | | -1 | -1 | -2 |
| 3 | 0 | 0 | 0 | 0 | 0 |
| 4 | 1 | 1 | 1 | 1 | 1 |
| 5 | 1.5 | | 2 | 2 | 1 |
| 6 | 2.5 | | 3 | 3 | 2 |
| 7 | 3.9 | | 4 | 4 | 3 |
| 8 | 4 | 0 | 4 | 4 | 4 |
| 9 | 4.1 | | 4 | 5 | 4 |
| 10 | 11 | 1 | 11 | 11 | 11 |

# EXAMPLE - Change data type transformation

This example illustrates how to clean up data by changing its data type to String, manipulating it using String functions, and then retyping the data to its proper data type.

**Functions:**

| Item | Description |
|------|-------------|
| IF Function | The `IF` function allows you to build if/then/else conditional logic within your transforms. |
| LEN Function | Returns the number of characters in a specified string. String value can be a column reference or string literal. |
| MERGE Function | Merges two or more columns of String type to generate output of String type. Optionally, you can insert a delimiter between the merged values. |

**Source:**

The following example contains customer ID and Zip code information in two columns. When this data is loaded into the Transformer page, it is initially interpreted as numeric, since it contains all numerals.

The four-digit `ZipCode` values should have five digits, with a `0` in front.

| CustId | ZipCode |
|--------|---------|
| 4020123 | 1234 |
| 2012121 | 94105 |
| 3212012 | 94101 |
| 1301212 | 2020 |

**Transformation:**

**CustId column:** This column needs to be retyped as String values. You can set the column data type to String through the column drop-down, which is rendered as the following transformation:

| Transformation Name | Change column data type |
|---------------------|-------------------------|
| Parameter: Columns | CustId |
| Parameter: New type | String |

While the column is now of String type, future transformations might cause it to be re-inferred as Integer values. To protect against this possibility, you might want to add a marker at the front of the string. This marker should be removed prior to execution.

The basic method is to create a new column containing the customer ID marker (`C`) and then merge this column and the existing `CustId` column together. It's useful to add such an indicator to the front in case the customer identifier is a numeric value that could be confused with other numeric values. Also, this merge step forces the value to be interpreted as a String value, which is more appropriate for an identifier.

| Transformation Name | Merge columns |
|---------------------|---------------|
| Parameter: Columns | 'C',CustId |

You can now delete the `CustId` columns and rename the new column as `CustId`.

**ZipCode column:** This column needs to be converted to valid Zip Code values. For ease of use, this column should be of type String:

| | |
|---|---|
| **Transformation Name** | `Change column data type` |
| **Parameter: Columns** | `ZipCode` |
| **Parameter: New type** | `Zipcode` |

The transformation below changes the value in the `ZipCode` column if the length of the value is four in any row. The new value is the original value prepended with the numeral `0`:

| | |
|---|---|
| **Transformation Name** | `Edit column with formula` |
| **Parameter: Columns** | `ZipCode` |
| **Parameter: Formula** | `if(len($col) == 4, merge(['0',$col]), $col)` |

This column might now be re-typed as Zipcode type.

**Results:**

| CustId | ZipCode |
|---|---|
| C4020123 | 01234 |
| C2012121 | 94105 |
| C3212012 | 94101 |
| C1301212 | 02020 |

Remember to remove the `C` marker from the `CustId` column. Select the `C` value in the `CustId` column and choose the `replace` transform. You might need to re-type the cleaned data as String data.

# EXAMPLE - Rename columns with rows

This example illustrates how you can rename columns based on the contents of specified rows.

**Source:**

You have imported the following racer data on heat times from a CSV file. When loaded in the Transformer page, it looks like the following:

| (rowId) | column2 | column3 | column4 | column5 |
|---------|---------|---------|---------|---------|
| 1 | Racer | Heat 1 | Heat 2 | Heat 3 |
| 2 | Racer X | 37.22 | 38.22 | 37.61 |
| 3 | Racer Y | 41.33 | DQ | 38.04 |
| 4 | Racer Z | 39.27 | 39.04 | 38.85 |

In the above, the `(rowId)` column references the row numbers displayed in the data grid; it is not part of the dataset. This information is available when you hover over the black dot on the left side of the screen.

**Transformation:**

You have examined the best performance in each heat according to the sample. You then notice that the data contains headers, but you forget how it was originally sorted. The data now looks like the following:

| (rowId) | column2 | column3 | column4 | column5 |
|---------|---------|---------|---------|---------|
| 1 | Racer Y | 41.33 | DQ | 38.04 |
| 2 | Racer | Heat 1 | Heat 2 | Heat 3 |
| 3 | Racer X | 37.22 | 38.22 | 37.61 |
| 4 | Racer Z | 39.27 | 39.04 | 38.85 |

You can use the following transformation to use the third row as your header for each column:

| | |
|---|---|
| **Transformation Name** | `Rename column with row(s)` |
| **Parameter: Option** | `Use row(s) as column names` |
| **Parameter: Type** | `Use a single row to name columns` |
| **Parameter: Row number** | 3 |

**Results:**

After you have applied the above transformation, your data should look like the following:

| (rowId) | Racer | Heat_1 | Heat_2 | Heat_3 |
|---------|-------|--------|--------|--------|
| 3 | Racer Y | 41.33 | DQ | 38.04 |
| 2 | Racer X | 37.22 | 38.22 | 37.61 |
| 4 | Racer Z | 39.27 | 39.04 | 38.85 |

# EXAMPLE - Splitting with Different Delimiter Types

This example shows how you can split data from a single column into multiple columns using delimiters.

- **single-pattern delimiter:** One pattern is applied one or more times to the source column to define the delimiters for the output columns
- **multi-pattern delimiter:** Multiple patterns, in the form of explicit strings, character index positions, or fixed-width fields, are used to split the column.

**Source:**

In this example, your CSV dataset contains status messages from a set of servers. In this case, the data about the server and the timestamp is contained in a single value within the CSV.

```
Server|Date Time,Status
admin.examplecom|2016-03-05 07:04:00,down
webapp.examplecom|2016-03-05 07:04:00,ok
admin.examplecom|2016-03-05 07:04:30,rebooting
webapp.examplecom|2016-03-05 07:04:00,ok
admin.examplecom|2016-03-05 07:05:00,ok
webapp.examplecom|2016-03-05 07:05:00,ok
```

**Transformation:**

When the data is first loaded into the Transformer page, the CSV data is split using the following two transformations:

| | |
|---|---|
| **Transformation Name** | `Split into rows` |
| **Parameter: Column** | `column1` |
| **Parameter: Split on** | `\n` |

| | |
|---|---|
| **Transformation Name** | `Split column` |
| **Parameter: Column** | `column1` |
| **Parameter: Option** | `On pattern` |
| **Parameter: Match pattern** | `','` |
| **Parameter: Ignore matches between** | `\"` |

You might need to add a `header` as the first step:

| | |
|---|---|
| **Transformation Name** | `Rename column with row(s)` |
| **Parameter: Option** | `Use row(s) as column names` |
| **Parameter: Type** | `Use a single row to name columns` |
| **Parameter: Row number** | `1` |

At this point, your data should look like the following:

| Server_Date_Time | Status |
|---|---|
| admin.example.com\|2016-03-05 07:04:00 | down |

| | |
|---|---|
| webapp.example.com\|2016-03-05 07:04:00 | ok |
| admin.example.com\|2016-03-05 07:04:30 | rebooting |
| webapp.example.com\|2016-03-05 07:04:30 | ok |
| admin.example.com\|2016-03-05 07:05:00 | ok |
| webapp.example.com\|2016-03-05 07:05:00 | ok |

The first column contains three distinct sets of data: the server name, the date, and the time. Note that the delimiters between these fields are different, so you should use a multi-pattern delimiter to break them apart:

| | |
|---|---|
| **Transformation Name** | `Split column` |
| **Parameter: Column** | `Server|Date Time` |
| **Parameter: Option** | `Sequence of patterns` |
| **Parameter: Pattern1** | `','` |
| **Parameter: Pattern2** | `' '` |

When the above is added, you should see three separate columns with the individual fields of information. Note that the source column has been automatically dropped.

Now, you decide that it would be useful to break apart the date information column into separate columns for year, month, and day. Since the column delimiter of this field is consistently a dash (–), you can use a single-pattern delimiter with the following transformation:

| | |
|---|---|
| **Transformation Name** | `Split by delimiter` |
| **Parameter: Column** | `Server|Date Time2` |
| **Parameter: Option** | `By delimiter` |
| **Parameter: Delimiter** | `'-'` |
| **Parameter: Number of columns to create** | `2` |

**Results:**

After you rename the generated columns, your dataset should look like the following. Note that the source timestamp column has been automatically dropped.

| server | year | month | day | time | Status |
|---|---|---|---|---|---|
| admin.example.com | 2016 | 03 | 05 | 07:04:00 | down |
| webapp.example.com | 2016 | 03 | 05 | 07:04:00 | ok |
| admin.example.com | 2016 | 03 | 05 | 07:04:30 | rebooting |
| webapp.example.com | 2016 | 03 | 05 | 07:04:30 | ok |
| admin.example.com | 2016 | 03 | 05 | 07:05:00 | ok |
| webapp.example.com | 2016 | 03 | 05 | 07:05:00 | ok |

# EXAMPLE - STARTSWITH and ENDSWITH Functions

This example demonstrates functions that can be used to evaluate the beginning and end of values of any type using patterns.

**Functions**:

| Item | Description |
|---|---|
| STARTSWIT H Function | Returns `true` if the leftmost set of characters of a column of values matches a pattern. The source value can be any data type, and the pattern can be a Pattern , regular expression, or a string. |
| ENDSWITH Function | Returns `true` if the rightmost set of characters of a column of values matches a pattern. The source value can be any data type, and the pattern can be a Pattern , regular expression, or a string. |

**Source:**

The following inventory report indicates available quantities of product by product name. You need to verify that the product names are valid according to the following rules:

- A product name must begin with a three-digit numeric brand identifier, followed by a dash.
- A product name must end with a dash, followed by a six-digit numeric SKU.

Source data looks like the following, with the Validation column having no values in it.

| InvDate | ProductName | Qty | Validation |
|---|---|---|---|
| 04/21/2017 | 412-Widgets-012345 | 23 | |
| 04/21/2017 | 04-Fidgets-120341 | 66 | |
| 04/21/2017 | 204-Midgets-4421 | 31 | |
| 04/21/2017 | 593-Gidgets-402012 | 24 | |

**Transformation:**

In this case, you must evaluate the `ProductName` column for two conditions. These conditional functions are the following:

```
IF(STARTSWITH(ProductName, `#{3}-`), 'Ok', 'Bad ProductName-Brand')
```

```
IF(ENDSWITH(ProductName, `-#{6}`), 'Ok', 'Bad ProductName-SKU')
```

One approach is to create two new test columns and then edit the column based on the evaluation of these two columns. However, using the following, you can compress the evaluation into a single step without creating the intermediate columns:

| | |
|---|---|
| **Transformation Name** | `Edit column with formula` |
| **Parameter: Columns** | `Status` |
| **Parameter: Formula** | `IF(STARTSWITH(ProductName, `#{3}-`), IF(ENDSWITH(ProductName, `-#{6}`), 'Ok', 'Bad ProductName-SKU'), 'Bad ProductName-Brand')` |

**Results:**

| InvDate | ProductName | Qty | Validation |
|---|---|---|---|
| 04/21/2017 | 412-Widgets-012345 | 23 | Ok |
| 04/21/2017 | 04-Fidgets-120341 | 66 | Bad ProductName-Brand |
| 04/21/2017 | 204-Midgets-4421 | 31 | Bad ProductName-SKU |
| 04/21/2017 | 593-Gidgets-402012 | 24 | Ok |

# EXAMPLE - Statistical Functions

This example illustrates how you can apply statistical functions to your dataset. Calculations include average (mean), max, min, standard deviation, and variance.

**Functions:**

| Item | Description |
|---|---|
| AVERAGE Function | Computes the average (mean) from all row values in a column or group. Input column can be of Integer or Decimal. |
| MIN Function | Computes the minimum value found in all row values in a column. Input column can be of Integer, Decimal or Datetime. |
| MAX Function | Computes the maximum value found in all row values in a column. Inputs can be Integer, Decimal, or Datetime. |
| VAR Function | Computes the variance among all values in a column. Input column can be of Integer or Decimal. If no numeric values are detected in the input column, the function returns 0. |
| STDEV Function | Computes the standard deviation across all column values of Integer or Decimal type. |
| NUMFORMAT Function | Formats a numeric set of values according to the specified number formatting. Source values can be a literal numeric value, a function returning a numeric value, or reference to a column containing an Integer or Decimal values. |
| MODE Function | Computes the mode (most frequent value) from all row values in a column, according to their grouping. Input column can be of Integer, Decimal, or Datetime type. |

**Source:**

Students took a test and recorded the following scores. You want to perform some statistical analysis on them:

| Student | Score |
|---|---|
| Anna | 84 |
| Ben | 71 |
| Caleb | 76 |
| Danielle | 87 |
| Evan | 85 |
| Faith | 92 |
| Gabe | 85 |
| Hannah | 99 |
| Ian | 73 |
| Jane | 68 |

**Transformation:**

You can use the following transformations to calculate the average (mean), minimum, and maximum scores:

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | AVERAGE(Score) |
| Parameter: New column | 'avgScore' |

| | |
|---|---|
| **name** | |

| | |
|---|---|
| **Transformation Name** | New formula |
| **Parameter: Formula type** | Single row formula |
| **Parameter: Formula** | MIN(Score) |
| **Parameter: New column name** | 'minScore' |

| | |
|---|---|
| **Transformation Name** | New formula |
| **Parameter: Formula type** | Single row formula |
| **Parameter: Formula** | MAX(Score) |
| **Parameter: New column name** | 'maxScore' |

To apply statistical functions to your data, you can use the VAR and STDEV functions, which can be used as the basis for other statistical calculations.

| | |
|---|---|
| **Transformation Name** | New formula |
| **Parameter: Formula type** | Single row formula |
| **Parameter: Formula** | VAR(Score) |
| **Parameter: New column name** | var_Score |

| | |
|---|---|
| **Transformation Name** | New formula |
| **Parameter: Formula type** | Single row formula |
| **Parameter: Formula** | STDEV(Score) |
| **Parameter: New column name** | stdev_Score |

For each score, you can now calculate the variation of each one from the average, using the following:

| | |
|---|---|
| **Transformation Name** | New formula |
| **Parameter: Formula type** | Single row formula |
| **Parameter: Formula** | ((Score - avg_Score) / stdev_Score) |
| **Parameter: New column name** | 'stDevs' |

Now, you want to apply grades based on a formula:

| Grade | standard deviations from avg (stDevs) |
|---|---|
| A | stDevs > 1 |
| B | stDevs > 0.5 |
| C | -1 <= stDevs <= 0.5 |

| D | stDevs < -1 |
|---|---|
| F | stDevs < -2 |

You can build the following transformation using the `IF` function to calculate grades.

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | IF((stDevs > 1),'A',IF((stDevs < -2),'F',IF((stDevs < -1),'D',IF((stDevs > 0.5),'B','C')))) |

To clean up the content, you might want to apply some formatting to the score columns. The following reformats the `stdev_Score` and `stDevs` columns to display two decimal places:

| Transformation Name | Edit column with formula |
|---|---|
| Parameter: Columns | stdev_Score |
| Parameter: Formula | NUMFORMAT(stdev_Score, '##.00') |

| Transformation Name | Edit column with formula |
|---|---|
| Parameter: Columns | stDevs |
| Parameter: Formula | NUMFORMAT(stDevs, '##.00') |

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | MODE(Score) |
| Parameter: New column name | 'modeScore' |

**Results:**

| Student | Score | modeScore | avgScore | minScore | maxScore | var_Score | stdev_Score | stDevs | Grade |
|---|---|---|---|---|---|---|---|---|---|
| Anna | 84 | 85 | 82 | 68 | 99 | 87.00000000000001 | 9.33 | 0.21 | C |
| Ben | 71 | 85 | 82 | 68 | 99 | 87.00000000000001 | 9.33 | -1.18 | D |
| Caleb | 76 | 85 | 82 | 68 | 99 | 87.00000000000001 | 9.33 | -0.64 | C |
| Danielle | 87 | 85 | 82 | 68 | 99 | 87.00000000000001 | 9.33 | 0.54 | B |
| Evan | 85 | 85 | 82 | 68 | 99 | 87.00000000000001 | 9.33 | 0.32 | C |
| Faith | 92 | 85 | 82 | 68 | 99 | 87.00000000000001 | 9.33 | 1.07 | A |
| Gabe | 85 | 85 | 82 | 68 | 99 | 87.00000000000001 | 9.33 | 0.32 | C |
| Hannah | 99 | 85 | 82 | 68 | 99 | 87.000000000 | 9.33 | 1.82 | A |

| | | | | | 00001 | | | |
|------|----|----|----|----|----|------------------------|------|-------|---|
| Ian  | 73 | 85 | 82 | 68 | 99 | 87.000000000<br>00001 | 9.33 | -0.96 | C |
| Jane | 68 | 85 | 82 | 68 | 99 | 87.000000000<br>00001 | 9.33 | -1.50 | D |

# EXAMPLE - Statistical Functions Sample Method

This example shows some of the statistical functions that use the sample method of computation.

**Functions**:

| Item | Description |
|------|-------------|
| STDEVSAMP Function | Computes the standard deviation across column values of Integer or Decimal type using the sample statistical method. |
| VARSAMP Function | Computes the variance among all values in a column using the sample statistical method. Input column can be of Integer or Decimal. If no numeric values are detected in the input column, the function returns 0. |
| STDEVSAMP IF Function | Generates the standard deviation of values by group in a column that meet a specific condition using the sample statistical method. |
| VARSAMPIF Function | Generates the variance of values by group in a column that meet a specific condition using the sample statistical method. |
| ROUND Function | Rounds input value to the nearest integer. Input can be an Integer, a Decimal, a column reference, or an expression. Optional second argument can be used to specify the number of digits to which to round. |

**Source:**

Students took tests on three consecutive Saturdays:

| Student | Date | Score |
|---------|------|-------|
| Andrew | 11/9/19 | 81 |
| Bella | 11/9/19 | 84 |
| Christina | 11/9/19 | 79 |
| David | 11/9/19 | 64 |
| Ellen | 11/9/19 | 61 |
| Fred | 11/9/19 | 63 |
| Andrew | 11/16/19 | 73 |
| Bella | 11/16/19 | 88 |
| Christina | 11/16/19 | 78 |
| David | 11/16/19 | 67 |
| Ellen | 11/16/19 | 87 |
| Fred | 11/16/19 | 90 |
| Andrew | 11/23/19 | 76 |
| Bella | 11/23/19 | 93 |
| Christina | 11/23/19 | 81 |
| David | 11/23/19 | 97 |
| Ellen | 11/23/19 | 97 |
| Fred | 11/23/19 | 91 |

**Transformation:**

You can use the following transformations to calculate standard deviation and variance across all dates using the sample method. Each computation has been rounded to three digits.

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | round(stdevsamp(Score), 3) |
| Parameter: New column name | 'stdevSamp' |

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | round(varsamp(Score), 3) |
| Parameter: New column name | 'varSamp' |

You can use the following to limit the previous statistical computations to the last two Saturdays of testing:

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | round(stdevsampif(Score, Date != '11\/9\/2019'), 3) |
| Parameter: New column name | 'stdevSampIf' |

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | round(varsampif(Score, Date != '11\/9\/2019'), 3) |
| Parameter: New column name | 'varSampIf' |

**Results:**

| Student | Date | Score | varSampIf | stdevSampIf | varSamp | stdevSamp |
|---|---|---|---|---|---|---|
| Andrew | 11/9/19 | 81 | 94.515 | 9.722 | 131.673 | 11.475 |
| Bella | 11/9/19 | 84 | 94.515 | 9.722 | 131.673 | 11.475 |
| Christina | 11/9/19 | 79 | 94.515 | 9.722 | 131.673 | 11.475 |
| David | 11/9/19 | 64 | 94.515 | 9.722 | 131.673 | 11.475 |
| Ellen | 11/9/19 | 61 | 94.515 | 9.722 | 131.673 | 11.475 |
| Fred | 11/9/19 | 63 | 94.515 | 9.722 | 131.673 | 11.475 |
| Andrew | 11/16/19 | 73 | 94.515 | 9.722 | 131.673 | 11.475 |
| Bella | 11/16/19 | 88 | 94.515 | 9.722 | 131.673 | 11.475 |
| Christina | 11/16/19 | 78 | 94.515 | 9.722 | 131.673 | 11.475 |
| David | 11/16/19 | 67 | 94.515 | 9.722 | 131.673 | 11.475 |

| | | | | | | |
|---|---|---|---|---|---|---|
| Ellen | 11/16/19 | 87 | 94.515 | 9.722 | 131.673 | 11.475 |
| Fred | 11/16/19 | 90 | 94.515 | 9.722 | 131.673 | 11.475 |
| Andrew | 11/23/19 | 76 | 94.515 | 9.722 | 131.673 | 11.475 |
| Bella | 11/23/19 | 93 | 94.515 | 9.722 | 131.673 | 11.475 |
| Christina | 11/23/19 | 81 | 94.515 | 9.722 | 131.673 | 11.475 |
| David | 11/23/19 | 97 | 94.515 | 9.722 | 131.673 | 11.475 |
| Ellen | 11/23/19 | 97 | 94.515 | 9.722 | 131.673 | 11.475 |
| Fred | 11/23/19 | 91 | 94.515 | 9.722 | 131.673 | 11.475 |

# EXAMPLE - String Cleanup Functions

This example demonstrates functions that can be used to clean up strings.

**Functions**:

| Item | Description |
|------|-------------|
| TRIM Function | Removes leading and trailing whitespace from a string. Spacing between words is not removed. |
| REMOVEWHITESPACE Function | Removes all whitespace from a string, including leading and trailing whitespace and all whitespace within the string. |
| REMOVESYMBOLS Function | Removes all characters from a string that are not letters, numbers, accented Latin characters, or whitespace. |

## Source:

In the following `(space)` and `(tab)` indicate space keys and tabs, respectively. Carriage return and newline characters are also supported by whitespace functions.

| Strings | source |
|---------|--------|
| String01 | this source(space)(space) |
| String02 | (tab)(tab)this source |
| String03 | (tab)(tab)this source(space)(space) |
| String04 | this source's? |
| String05 | Why, you @#$%^&*()! |
| String06 | this sörce |
| String07 | (space)this sörce |
| String08 | à mañana |

## Transformation:

The following transformation steps generate new columns using each of the string cleanup functions:

| Transformation Name | New formula |
|---------------------|-------------|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | TRIM(source) |
| Parameter: New column name | 'trim_source' |

| Transformation Name | New formula |
|---------------------|-------------|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | REMOVEWHITESPACE(source) |
| Parameter: New column name | 'removewhitespace_source' |

| Transformation Name | New formula |
|---------------------|-------------|

| Parameter: Formula type | Single row formula |
|---|---|
| Parameter: Formula | REMOVESYMBOLS(source) |
| Parameter: New column name | 'removesymbols_source' |

**Results:**

| Strings | source | removesymbols_source | removewhitespace_source | trim_source |
|---|---|---|---|---|
| String01 | this source(space)(space) | this source(space)(space) | thissource | this source |
| String02 | (tab)(tab)this source | (tab)(tab)this source | thissource | this source |
| String03 | (tab)(tab)this source(space)(space) | (tab)(tab)this source(space)(space) | thissource | this source |
| String04 | this source's? | this sources | thissource's? | this source's? |
| String05 | "Why, you @#$%^&*()!" | Why you | Why,you@#$%^&*()! | Why, you @#$%^&*()! |
| String06 | this söurce | this söurce | thissöurce | this söurce |
| String07 | (space)this söurce | (space)this söurce | thissöurce | this söurce |
| String08 | à mañana | à mañana | àmañana | à  mañana |

# EXAMPLE - String Comparison Functions

This example demonstrates functions that can be used to compare two sets of strings.

**Functions**:

| Item | Description |
|---|---|
| STRINGGREATERTHAN Function | Returns `true` if the first string evaluates to be greater than the second string, based on a set of common collation rules. |
| STRINGGREATERTHAN EQUAL Function | Returns `true` if the first string evaluates to be greater than or equal to the second string, based on a set of common collation rules. |
| STRINGLESSTHAN Function | Returns `true` if the first string evaluates to be less than the second string, based on a set of common collation rules. |
| STRINGLESSTHANEQU AL Function | Returns `true` if the first string evaluates to be less than or equal to the second string, based on a set of common collation rules. |
| EXACT Function | Returns `true` if the second string evaluates to be an exact match of the first string. Source values can be string literals, column references, or expressions that evaluate to strings. |

**Source:**

The following table contains some example strings to be compared.

| rowId | stringA | stringB |
|---|---|---|
| 1 | a | a |
| 2 | a | A |
| 3 | a | b |
| 4 | a | 1 |
| 5 | a | ; |
| 6 | ; | 1 |
| 7 | a | a |
| 8 | a | aa |
| 9 | abc | x |

Note that in row #6, `stringB` begins with a space character.

**Transformation:**

For each set of strings, the following functions are applied to generate a new column containing the results of the comparison.

| Transformation Name | New formula |
|---|---|
| **Parameter: Formula type** | Single row formula |
| **Parameter: Formula** | STRINGGREATERTHAN(stringA,stringB) |
| **Parameter: New column name** | 'greaterThan' |

| Transformation Name | New formula |
| --- | --- |
| Parameter: Formula type | Single row formula |
| Parameter: Formula | STRINGGREATERTHANEQUAL(stringA,stringB) |
| Parameter: New column name | 'greaterThanEqual' |

| Transformation Name | New formula |
| --- | --- |
| Parameter: Formula type | Single row formula |
| Parameter: Formula | STRINGLESSTHAN(stringA,stringB) |
| Parameter: New column name | 'lessThan' |

| Transformation Name | New formula |
| --- | --- |
| Parameter: Formula type | Single row formula |
| Parameter: Formula | STRINGLESSTHANEQUAL(stringA,stringB) |
| Parameter: New column name | 'lessThanEqual' |

| Transformation Name | New formula |
| --- | --- |
| Parameter: Formula type | Single row formula |
| Parameter: Formula | EXACT(stringA,stringB) |
| Parameter: New column name | 'exactEqual' |

**Results:**

In the following table, the Notes column has been added manually.

| rowId | stringA | stringB | lessThanEqual | lessThan | greaterThanEqual | greaterThan | exactEqual | Notes |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 1 | a | a | true | false | true | false | true | Evaluation of differences between STRINGLESSTHAN and STRINGGREATERTHAN and greater than versions. |
| 2 | a | A | true | true | false | false | false | Comparisons are case-sensitive. Uppercase letters are |

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| | | | | | | | greater than lowercase letters. |
| 3 | a | b | true | true | false | false | false | Letters later in the alphabet (b) are greater than earlier letters (a). |
| 4 | a | 1 | false | false | true | true | false | Letters (a) are greater than digits (1). |
| 5 | a | ; | false | false | true | true | false | Letters (a) are greater than non-alphanumerics (;). |
| 6 | ; | 1 | true | true | false | false | false | Digits (1) are greater than non-alphanumerics (;). Therefore, the following characters are listed in order of evaluation:<br><br>`Aa1;` |
| 7 | a |   | false | false | true | true | false | Letters (and any non-breaking character) are greater than space values. |
| 8 | a | aa | true | true | false | false | false | The second string is greater, since it contains one additional string at the end. |
| 9 | abc | x | true | true | false | false | false | The second string is greater, since its first letter is greater than the first letter of the first string. |

# EXAMPLE - SUMIF and COUNTDISTINCTIF Functions

This example illustrates how you can use conditional calculation functions.

**Functions**:

| Item | Description |
|---|---|
| SUMIF Function | Generates the sum of rows in each group that meet a specific condition. |
| COUNTDISTINCTIF Function | Generates the count of distinct non-null values for rows in each group that meet a specific condition. |

**Source:**

Here is some example polling data across 16 precincts in 8 cities across 4 counties, where registrations have been invalidated at the polling station, preventing voters from voting. Precincts where this issue has occurred previously have been added to a watch list (`precinctWatchList`).

| totalReg | invalidReg | precinctWatchList | precinctId | cityId | countyId |
|---|---|---|---|---|---|
| 731 | 24 | y | 1 | 1 | 1 |
| 743 | 29 | y | 2 | 1 | 1 |
| 874 | 0 | | 3 | 2 | 1 |
| 983 | 0 | | 4 | 2 | 1 |
| 622 | 29 | | 5 | 3 | 2 |
| 693 | 0 | | 6 | 3 | 2 |
| 775 | 37 | y | 7 | 4 | 2 |
| 1025 | 49 | y | 8 | 4 | 2 |
| 787 | 13 | | 9 | 5 | 3 |
| 342 | 0 | | 10 | 5 | 3 |
| 342 | 39 | y | 11 | 6 | 3 |
| 387 | 28 | y | 12 | 6 | 3 |
| 582 | 59 | | 13 | 7 | 4 |
| 244 | 0 | | 14 | 7 | 4 |
| 940 | 6 | y | 15 | 8 | 4 |
| 901 | 4 | y | 16 | 8 | 4 |

**Transformation:**

First, you want to sum up the invalid registrations (`invalidReg`) for precincts that are already on the watchlist (`precinctWatchList = y`). These sums are grouped by city, which can span multiple precincts:

| Transformation Name | `New formula` |
|---|---|
| Parameter: Formula type | `Single row formula` |
| Parameter: Formula | `SUMIF(invalidReg, precinctWatchList == "y")` |
| Parameter: Group rows by | `cityId` |

| | |
|---|---|
| **Parameter: New column name** | `'invalidRegbyCityId'` |

The `invalidRegbyCityId` column contains invalid registrations across the entire city.

Now, at the county level, you want to identify the number of precincts that were on the watch list and were part of a city-wide registration problem.

In the following step, the number of cities in each count are counted where invalid registrations within a city is greater than `60`.

- This step creates a pivot aggregation.

| Transformation Name | `Pivot columns` |
|---|---|
| **Parameter: Row labels** | `countyId` |
| **Parameter: Values** | `COUNTDISTINCTIF(precinctId, invalidRegbyCityId > 60)` |
| **Parameter: Max number of columns to create** | `1` |

**Results:**

| countyId | countdistinctif_precinctId |
|---|---|
| 1 | 0 |
| 2 | 2 |
| 3 | 2 |
| 4 | 0 |

The voting officials in counties 2 and 3 should investigate their precinct registration issues.

# EXAMPLE - SUMIF Function

This example can be used to sum the values in a column based on a condition and organized by group.

**Function**:

| Item | Description |
|---|---|
| SUMIF Function | Generates the sum of rows in each group that meet a specific condition. |
| WEEKDAY Function | Derives the numeric value for the day of the week ($1$, $2$, etc.). Input must be a reference to a column containing Datetime values. |

## Source:

The following data identifies sales figures by salespeople for a week:

| EmployeeId | Date | Sales |
|---|---|---|
| S001 | 1/23/17 | 25 |
| S002 | 1/23/17 | 40 |
| S003 | 1/23/17 | 48 |
| S001 | 1/24/17 | 81 |
| S002 | 1/24/17 | 11 |
| S003 | 1/24/17 | 25 |
| S001 | 1/25/17 | 9 |
| S002 | 1/25/17 | 40 |
| S003 | 1/25/17 | |
| S001 | 1/26/17 | 77 |
| S002 | 1/26/17 | 83 |
| S003 | 1/26/17 | |
| S001 | 1/27/17 | 17 |
| S002 | 1/27/17 | 71 |
| S003 | 1/27/17 | 29 |
| S001 | 1/28/17 | |
| S002 | 1/28/17 | |
| S003 | 1/28/17 | 14 |
| S001 | 1/29/17 | 2 |
| S002 | 1/29/17 | 7 |
| S003 | 1/29/17 | 99 |

## Transformation:

You want to know how your salespeople are doing by the day of the week. To the above, you add a column that identifies the day of the week:

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | `Single row formula` |
| Parameter: Formula | `WEEKDAY(Date)` |
| Parameter: New column name | `'DayOfWeek'` |

First you wish to examine weekday sales, when `DayOfWeek < 6`. For each day of the week, you can preview the following aggregation:

| Transformation Name | `Pivot columns` |
|---|---|
| Parameter: Row labels | `groupId` |
| Parameter: Values | `sumif(Sales, DayOfWeek < 6)` |

Performance is listed in the following order: S001, S002, S003.

To analyze the weekend, you change the above to the following:

| Transformation Name | `Pivot columns` |
|---|---|
| Parameter: Row labels | `groupId` |
| Parameter: Values | `sumif(Sales, (DayOfWeek >= 5))` |

**Results:**

The following are the results for the weekend:

| EmployeeId | sumif_Sales |
|---|---|
| S001 | 42 |
| S002 | 126 |
| S003 | 142 |

# EXAMPLE - Time Zone Conversion Functions

This example shows how you can use functions to convert Datetime values to different time zones.

**Functions**:

| Item | Description |
|---|---|
| CONVERTFROM UTC Function | Converts Datetime value to corresponding value of the specified time zone. Input can be a column of Datetime values, a literal Datetime value, or a function returning Datetime values. |
| CONVERTTOUT C Function | Converts Datetime value in specified time zone to corresponding value in UTC time zone. Input can be a column of Datetime values, a literal Datetime value, or a function returning Datetime values. |
| CONVERTTIMEZ ONE Function | Converts Datetime value in specified time zone to corresponding value second specified time zone. Input can be a column of Datetime values, a literal Datetime value, or a function returning Datetime values. |
| ISMISMATCHED Function | Tests whether a set of values is not valid for a specified data type. |

**Source:**

| row | datetime |
|---|---|
| 1 | 2020-03-15 |
| 2 | 2020-03-15 0:00:00 |
| 3 | 2020-03-15 +08:00 |
| 4 | 2020-03-15 1:02:03 |
| 5 | 2020-03-15 4:02:03 |
| 6 | 2020-03-15 8:02:03 |
| 7 | 2020-03-15 12:02:03 |
| 8 | 2020-03-15 16:02:03 |
| 9 | 2020-03-15 20:02:03 |
| 10 | 2020-03-15 23:02:03 |

**Transformation:**

When you import the above dates, Designer Cloud Powered by Trifacta Enterprise Edition may not recognize the column as a set of dates. You can use the column menus to format the date values to the following standardized format:

```
yyyy*mm*dd*HH:MM:SS
```

| Transformation Name | Change column data type |
|---|---|
| Parameter: Columns | datetime |
| Parameter: New type | Date/Time |
| Parameter: Date/Time type | yyyy*mm*dd*HH:MM:SS |

When the type has been changed, row 1 and row 3 have been identified as invalid. You can use the following transformation to remove these rows:

| Transformation Name | Filter rows |
|---|---|
| Parameter: Condition | Custom formula |
| Parameter: Type of formula | Custom single |
| Parameter: Condition | ISMISMATCHED(datetime, ['Datetime','yy-mm-dd hh:mm:ss','yyyy*mm*dd*HH:MM:SS']) |
| Parameter: Action | Delete matching rows |

When the Datetime values are consistently formatted, you can use the following transformations to perform conversions. The following tranformation converts the values from UTC to US/Eastern time zone:

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | CONVERTFROMUTC(datetime, 'US\/Eastern') |
| Parameter: New column name | 'datetimeUTC2Eastern' |

This transformation now assumes that the date values are in US/Pacific time zone and converts them to UTC:

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | CONVERTTOUTC(datetime, 'US\/Pacific') |
| Parameter: New column name | 'datetimePacific2UTC' |

The final transformation converts the date time values between arbitrary time zones. In this case, the values are assumed to be in US/Alaska time zone and are converted to US/Hawaii time zone:

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | CONVERTTIMEZONE(datetime, 'US\/Alaska', 'US\/Hawaii') |
| Parameter: New column name | 'datetimeAlaska2Hawaii' |

**Results:**

| row | datetime | datetimeAlaska2Hawaii | datetimePacific2UTC | datetimeUTC2Eastern |
|---|---|---|---|---|
| 2 | 2020-03-15 00:00:00 | 2020-03-14 22:00:00 | 2020-03-15 07:00:00 | 2020-03-14 20:00:00 |
| 4 | 2020-03-15 01:02:03 | 2020-03-14 23:02:03 | 2020-03-15 08:02:03 | 2020-03-14 21:02:03 |
| 5 | 2020-03-15 04:02:03 | 2020-03-15 02:02:03 | 2020-03-15 11:02:03 | 2020-03-15 00:02:03 |
| 6 | 2020-03-15 08:02:03 | 2020-03-15 06:02:03 | 2020-03-15 15:02:03 | 2020-03-15 04:02:03 |
| 7 | 2020-03-15 12:02:03 | 2020-03-15 10:02:03 | 2020-03-15 19:02:03 | 2020-03-15 08:02:03 |
| 8 | 2020-03-15 16:02:03 | 2020-03-15 14:02:03 | 2020-03-15 23:02:03 | 2020-03-15 12:02:03 |

| 9 | 2020-03-15 20:02:03 | 2020-03-15 18:02:03 | 2020-03-16 03:02:03 | 2020-03-15 16:02:03 |
| 10 | 2020-03-15 23:02:03 | 2020-03-15 21:02:03 | 2020-03-16 06:02:03 | 2020-03-15 19:02:03 |

# EXAMPLE - Trigonometry Arc Functions

This example illustrates how to apply the inverse trigonometric (Arc) functions to your transformations.

**Functions:**

| Item | Description |
|---|---|
| ASIN Function | For input values between -1 and 1 inclusive, this function returns the angle in radians whose sine value is the input. This function is the inverse of the sine function. The value can be a Decimal or Integer literal or a reference to a column containing numeric values. |
| ACOS Function | For input values between -1 and 1 inclusive, this function returns the angle in radians whose cosine value is the input. This function is the inverse of the cosine function. The value can be a Decimal or Integer literal or a reference to a column containing numeric values. |
| ATAN Function | For input values between -1 and 1 inclusive, this function returns the angle in radians whose tangent value is the input. This function is the inverse of the tangent function. The value can be a Decimal or Integer literal or a reference to a column containing numeric values. |

Also:

| Item | Description |
|---|---|
| ROUND Function | Rounds input value to the nearest integer. Input can be an Integer, a Decimal, a column reference, or an expression. Optional second argument can be used to specify the number of digits to which to round. |
| DEGREES Function | Computes the degrees of an input value measuring the radians of an angle. The value can be a Decimal or Integer literal or a reference to a column containing numeric values. |

> **NOTE:** These functions are valid over specific ranges.

The following functions are computed using the above functions.

- Arccotangent. Computed using ATAN function. See below.
- Arcsecant. Computed using ACOS function. See below.
- Arccosecant. Computed using ASIN function. See below.

**Source:**

In the following sample, input values are in radians. In this example, all values are rounded to two decimals for clarity.

| Y |
|---|
| -1.00 |
| -0.75 |
| -0.50 |
| 0.00 |
| 0.50 |
| 0.75 |
| 1.00 |

**Transformation:**

Arcsine:

Valid over the range (-1 <= Y <= 1)

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | round(degrees(asin(Y)), 2) |
| Parameter: New column name | 'asinY' |

Arccosine:

Valid over the range (-1 <= Y <= 1)

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | round(degrees(acos(Y)), 2) |
| Parameter: New column name | 'acosY' |

Arctangent:

Valid over the range (-1 <= Y <= 1)

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | round(degrees(atan(Y)), 2) |
| Parameter: New column name | 'atanY' |

Arccosecant:

This function is valid over a set of ranged inputs, so you can use a conditional column for the computation.

| Transformation Name | Conditional column |
|---|---|
| Parameter: Condition type | if...then...else |
| Parameter: If | (Y <= -1) \|\| (Y >= 1) |
| Parameter: Then | round(degrees(asin(divide(1, Y))), 2) |
| Parameter: New column name | 'acscY' |

Arcsecant:

Same set of ranged inputs apply to this function.

| Transformation Name | `Conditional column` |
|---|---|
| **Parameter: Condition type** | `if...then...else` |
| **Parameter: If** | `(Y <= -1) \|\| (Y >= 1)` |
| **Parameter: Then** | `round(degrees(acos(divide(1, Y))), 2)` |
| **Parameter: New column name** | `'asecY'` |

Arccotangent:

For this function, the two different ranges of inputs have different computations, so an `else` condition is added to the transformation.

| Transformation Name | `Conditional column` |
|---|---|
| **Parameter: Condition type** | `if...then...else` |
| **Parameter: If** | `Y > 0` |
| **Parameter: Then** | `round(degrees(atan(divide(1, Y))), 2)` |
| **Parameter: Else** | `round(degrees(atan(divide(1, Y)) + pi()), 2)` |
| **Parameter: New column name** | `'acotY'` |

**Results:**

| Y | acotY | asecY | acscY | atanY | acosY | asinY |
|---|---|---|---|---|---|---|
| -1.00 | -41.86 | 180.00 | -90.00 | -45.00 | 180.00 | -90.00 |
| -0.75 | -49.99 | *null* | *null* | -37.00 | 139.00 | -49.00 |
| -0.50 | -60.29 | *null* | *null* | -27.00 | 120.00 | -30.00 |
| 0.00 | *null* | *null* | *null* | 0.00 | 90.00 | 0.00 |
| 0.50 | 63.44 | *null* | *null* | 27.00 | 60.00 | 30.00 |
| 0.75 | 53.13 | *null* | *null* | 37.00 | 41.00 | 49.00 |
| 1.00 | 45.00 | 0.00 | 90.00 | 45.00 | 0.00 | 90.00 |

# EXAMPLE - Trigonometry Functions

This example illustrates how to apply basic trigonometric functions to your transformations.

**Functions**:

| Item | Description |
|------|-------------|
| SIN Function | Computes the sine of an input value for an angle measured in radians. The value can be a Decimal or Integer literal or a reference to a column containing numeric values. |
| COS Function | Computes the cosine of an input value for an angle measured in radians. The value can be a Decimal or Integer literal or a reference to a column containing numeric values. |
| TAN Function | Computes the tangent of an input value for an angle measured in radians. The value can be a Decimal or Integer literal or a reference to a column containing numeric values. |

The following functions are computed using the above functions:

- **Cotangent.** Computed as 1/TAN.
- **Secant.** Computed as 1/COS.
- **Cosecant.** Computed as 1/SIN.

Also:

| Item | Description |
|------|-------------|
| ROUND Function | Rounds input value to the nearest integer. Input can be an Integer, a Decimal, a column reference, or an expression. Optional second argument can be used to specify the number of digits to which to round. |
| RADIANS Function | Computes the radians of an input value measuring degrees of an angle. The value can be a Decimal or Integer literal or a reference to a column containing numeric values. |

**Source:**

In the following sample, input values are in degrees:

| X |
|---|
| -30 |
| 0 |
| 30 |
| 45 |
| 60 |
| 90 |
| 120 |
| 135 |
| 180 |

**Transformation:**

In this example, all values are rounded to three decimals for clarity.

First, the above values in degrees must be converted to radians.

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | round(radians(X), 3) |
| Parameter: New column name | 'rX' |

Sine:

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | round(sin(rX), 3) |
| Parameter: New column name | 'SINrX' |

Cosine:

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | round(cos(rX), 3) |
| Parameter: New column name | 'COSrX' |

Tangent:

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | round(tan(rX), 3) |
| Parameter: New column name | 'TANrX' |

Cotangent:

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | round(1 / tan(rX), 3) |
| Parameter: New column name | 'COTrX' |

Secant:

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | round(1 / cos(rX), 3) |

| Parameter: New column name | 'SECrX' |
|---|---|

Cosecant:

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | round(1 / sin(rX), 3) |
| Parameter: New column name | 'CSCrX' |

**Results:**

| X | rX | COTrX | SECrX | CSCrX | TANrX | COSrX | SINrX |
|---|---|---|---|---|---|---|---|
| -30 | -0.524 | -1.73 | 1.155 | -1.999 | -0.578 | 0.866 | -0.5 |
| 0 | 0 | *null* | 1 | *null* | 0 | 1 | 0 |
| 30 | 0.524 | 1.73 | 1.155 | 1.999 | 0.578 | 0.866 | 0.5 |
| 45 | 0.785 | 1.001 | 1.414 | 1.415 | 0.999 | 0.707 | 0.707 |
| 60 | 1.047 | 0.578 | 1.999 | 1.155 | 1.731 | 0.5 | 0.866 |
| 90 | 1.571 | 0 | -4909.826 | 1 | -4909.826 | 0 | 1 |
| 120 | 2.094 | -0.577 | -2.001 | 1.154 | -1.734 | -0.5 | 0.866 |
| 135 | 2.356 | -1 | -1.414 | 1.414 | -1 | -0.707 | 0.707 |
| 180 | 3.142 | 2454.913 | -1 | -2454.913 | 0 | -1 | 0 |

# EXAMPLE - Trigonometry Hyperbolic Arc Functions

This example illustrates how to apply inverse (arc) hyperbolic functions to your transformations.

**Functions:**

| Item | Description |
|------|-------------|
| ASINH Function | Computes the arcsine of an input value for a hyperbolic angle measured in radians. The value can be a Decimal or Integer literal or a reference to a column containing numeric values. |
| ACOSH Function | Computes the arccosine of an input value for a hyperbolic angle measured in radians. The value can be a Decimal or Integer literal or a reference to a column containing numeric values. |
| ATANH Function | Computes the arctangent of an input value for a hyperbolic angle measured in radians. The value can be a Decimal or Integer literal or a reference to a column containing numeric values. |

Also:

| Item | Description |
|------|-------------|
| ROUND Function | Rounds input value to the nearest integer. Input can be an Integer, a Decimal, a column reference, or an expression. Optional second argument can be used to specify the number of digits to which to round. |
| DEGREES Function | Computes the degrees of an input value measuring the radians of an angle. The value can be a Decimal or Integer literal or a reference to a column containing numeric values. |
| NULL Function | The NULL function generates null values. |

**Source:**

In the following sample, input values are in radians. In this example, all values are rounded to two decimals for clarity.

| Y |
|---|
| -4.00 |
| -3.00 |
| -2.00 |
| -1.00 |
| -0.75 |
| -0.50 |
| 0.00 |
| 0.50 |
| 0.75 |
| 1.00 |
| 2.00 |
| 3.00 |
| 4.00 |

**Transformation:**

The following transformations include checks for the valid ranges for input values.

Hyperbolic arcsine:

| Transformation Name | New formula |
| --- | --- |
| Parameter: Formula type | Single row formula |
| Parameter: Formula | round(degrees(asinh(Y)), 2) |
| Parameter: New column name | 'asinhY' |

Hyperbolic arccosine:

Valid over the range (y > 1)

| Transformation Name | New formula |
| --- | --- |
| Parameter: Formula type | Single row formula |
| Parameter: Formula | if(Y>1,round(degrees(acosh(Y)), 2),null()) |
| Parameter: New column name | 'acoshY' |

Hyperbolic arctangent:

Valid over the range (-1 < y < 1)

| Transformation Name | New formula |
| --- | --- |
| Parameter: Formula type | Single row formula |
| Parameter: Formula | if(abs(y)<1,round(degrees(atanh(Y)), 2),null()) |
| Parameter: New column name | 'atanhY' |

**Results:**

| Y | atanhY | acoshY | asinhY |
| --- | --- | --- | --- |
| -4 | null | null | -120.02 |
| -3 | null | null | -104.19 |
| -2 | null | null | -82.71 |
| -1.5 | null | null | -68.45 |
| -1 | null | null | -50.5 |
| -0.75 | -55.75 | null | -39.71 |
| -0.5 | -31.47 | null | -27.57 |
| 0 | 0 | null | 0 |
| 0.5 | 31.47 | null | 27.57 |
| 0.75 | 55.75 | null | 39.71 |
| 1 | null | null | 50.5 |
| 1.5 | null | 55.14 | 68.45 |

| 2 | *null* | 75.46 | 82.71 |
| 3 | *null* | 101 | 104.19 |
| 4 | *null* | 118.23 | 120.02 |

# EXAMPLE - Trigonometry Hyperbolic Functions

This example illustrates how to apply hyperbolic trigonometric functions to your transformations. All of the functions take inputs in radians.

**Functions:**

| Item | Description |
|------|-------------|
| SINH Function | Computes the hyperbolic sine of an input value for a hyperbolic angle measured in radians. The value can be a Decimal or Integer literal or a reference to a column containing numeric values. |
| COSH Function | Computes the hyperbolic cosine of an input value for a hyperbolic angle measured in radians. The value can be a Decimal or Integer literal or a reference to a column containing numeric values. |
| TANH Function | Computes the hyperbolic tangent of an input value for a hyperbolic angle measured in radians. The value can be a Decimal or Integer literal or a reference to a column containing numeric values. |

The following functions can be computed based on the inverse of the above functions:

- Hyperbolic Cotangent. Computed as 1/TANH.
- Hyperbolic Secant. Computed as 1/COSH.
- Hyperbolic Cosecant. Computed as 1/SINH.

Also:

| Item | Description |
|------|-------------|
| ROUND Function | Rounds input value to the nearest integer. Input can be an Integer, a Decimal, a column reference, or an expression. Optional second argument can be used to specify the number of digits to which to round. |
| RADIANS Function | Computes the radians of an input value measuring degrees of an angle. The value can be a Decimal or Integer literal or a reference to a column containing numeric values. |
| DIVIDE Function | Returns the value of dividing the first argument by the second argument. Equivalent to the / operator. |

**Source:**

In the following sample, input values are in degrees:

| X |
|---|
| -30 |
| 0 |
| 30 |
| 45 |
| 60 |
| 90 |
| 120 |
| 135 |
| 180 |

**Transformation:**

In this example, all values are rounded to three decimals for clarity.

First, the above values in degrees must be converted to radians.

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | round(radians(X), 3) |
| Parameter: New column name | 'rX' |

Hyperbolic Sine:

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | round(sinh(rX), 3) |
| Parameter: New column name | 'SINHrX' |

Hyperbolic Cosine:

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | round(cosh(rX), 3) |
| Parameter: New column name | 'COSHrX' |

Hyperbolic Tangent:

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | round(tanh(rX), 3) |
| Parameter: New column name | 'TANHrX' |

Hyperbolic Cotangent:

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | round(divide(1, tanh(rX)), 3) |
| Parameter: New column name | 'COTHrX' |

Hyperbolic Secant:

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |

| Parameter: Formula | `round(divide(1, cosh(rX)), 3)` |
|---|---|
| Parameter: New column name | `'SECHrX'` |

Hyperbolic Cosecant:

| Transformation Name | `New formula` |
|---|---|
| Parameter: Formula type | `Single row formula` |
| Parameter: Formula | `round(divide(1, sinh(rX)), 3)` |
| Parameter: New column name | `'CSCHrX'` |

**Results:**

| X | rX | TANHrX | COTHrX | COSHrX | SECHrX | SINHrX | CSCHrX |
|---|---|---|---|---|---|---|---|
| -30 | -0.524 | -0.481 | -2.079 | 1.14 | 0.877 | -0.548 | -1.825 |
| 0 | 0 | 0 | *null* | 1 | 1 | 0 | *null* |
| 30 | 0.524 | 0.481 | 2.079 | 1.14 | 0.877 | 0.548 | 1.825 |
| 45 | 0.785 | 0.656 | 1.524 | 1.324 | 0.755 | 0.868 | 1.152 |
| 60 | 1.047 | 0.781 | 1.28 | 1.6 | 0.625 | 1.249 | 0.801 |
| 90 | 1.571 | 0.917 | 1.091 | 2.51 | 0.398 | 2.302 | 0.434 |
| 120 | 2.094 | 0.97 | 1.031 | 4.12 | 0.243 | 3.997 | 0.25 |
| 135 | 2.356 | 0.982 | 1.018 | 5.322 | 0.188 | 5.227 | 0.191 |
| 180 | 3.142 | 0.996 | 1.004 | 11.597 | 0.086 | 11.553 | 0.087 |

# EXAMPLE - Two-Column Statistical Functions

This example illustrates statistical functions that can be applied across two columns of values.

**Functions**:

| Item | Description |
|---|---|
| CORREL Function | Computes the correlation coefficient between two columns. Source values can be of Integer or Decimal type. |
| COVAR Function | Computes the covariance between two columns using the population method. Source values can be of Integer or Decimal type. |
| COVARSAMP Function | Computes the covariance between two columns using the sample method. Source values can be of Integer or Decimal type. |
| ROUND Function | Rounds input value to the nearest integer. Input can be an Integer, a Decimal, a column reference, or an expression. Optional second argument can be used to specify the number of digits to which to round. |

**Source:**

The following table contains height in inches and weight in pounds for a set of students.

| Student | heightIn | weightLbs |
|---|---|---|
| 1 | 70 | 134 |
| 2 | 67 | 135 |
| 3 | 67 | 147 |
| 4 | 67 | 160 |
| 5 | 72 | 136 |
| 6 | 73 | 146 |
| 7 | 71 | 135 |
| 8 | 63 | 145 |
| 9 | 67 | 138 |
| 10 | 66 | 138 |
| 11 | 71 | 161 |
| 12 | 70 | 131 |
| 13 | 74 | 131 |
| 14 | 67 | 157 |
| 15 | 73 | 161 |
| 16 | 70 | 133 |
| 17 | 63 | 132 |
| 18 | 64 | 153 |
| 19 | 64 | 156 |
| 20 | 72 | 154 |

**Transformation:**

You can use the following transformations to calculate the correlation co-efficient, the covariance, and the sampling method covariance between the two data columns:

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | round(correl(heightIn, weightLbs), 3) |
| Parameter: New column name | 'corrHeightAndWeight' |

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | round(covar(heightIn, weightLbs), 3) |
| Parameter: New column name | 'covarHeightAndWeight' |

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | round(covarsamp(heightIn, weightLbs), 3) |
| Parameter: New column name | 'covarHeightAndWeight-Sample' |

**Results:**

| Student | heightIn | weightLbs | covarHeightAndWeight-Sample | covarHeightAndWeight | corrHeightAndWeight |
|---|---|---|---|---|---|
| 1 | 70 | 134 | -2.876 | -2.732 | -0.074 |
| 2 | 67 | 135 | -2.876 | -2.732 | -0.074 |
| 3 | 67 | 147 | -2.876 | -2.732 | -0.074 |
| 4 | 67 | 160 | -2.876 | -2.732 | -0.074 |
| 5 | 72 | 136 | -2.876 | -2.732 | -0.074 |
| 6 | 73 | 146 | -2.876 | -2.732 | -0.074 |
| 7 | 71 | 135 | -2.876 | -2.732 | -0.074 |
| 8 | 63 | 145 | -2.876 | -2.732 | -0.074 |
| 9 | 67 | 138 | -2.876 | -2.732 | -0.074 |
| 10 | 66 | 138 | -2.876 | -2.732 | -0.074 |
| 11 | 71 | 161 | -2.876 | -2.732 | -0.074 |
| 12 | 70 | 131 | -2.876 | -2.732 | -0.074 |
| 13 | 74 | 131 | -2.876 | -2.732 | -0.074 |
| 14 | 67 | 157 | -2.876 | -2.732 | -0.074 |
| 15 | 73 | 161 | -2.876 | -2.732 | -0.074 |
| 16 | 70 | 133 | -2.876 | -2.732 | -0.074 |
| 17 | 63 | 132 | -2.876 | -2.732 | -0.074 |
| 18 | 64 | 153 | -2.876 | -2.732 | -0.074 |

| 19 | 64 | 156 | -2.876 | -2.732 | -0.074 |
| 20 | 72 | 154 | -2.876 | -2.732 | -0.074 |

# EXAMPLE - Type Functions

This example illustrates how various type checking functions can be applied to your data.

**Functions**:

| Item | Description |
|------|-------------|
| VALID Function | Tests whether a set of values is valid for a specified data type and is not a null value. |
| ISMISMATCHED Function | Tests whether a set of values is not valid for a specified data type. |
| ISMISSING Function | The `ISMISSING` function tests whether a column of values is missing or null. For input column references, this function returns `true` or `false`. |
| ISNULL Function | The `ISNULL` function tests whether a column of values contains null values. For input column references, this function returns `true` or `false`. |
| NULL Function | The `NULL` function generates null values. |

**Source:**

Some source values that should match the State and Integer data types:

| State | Qty |
|-------|-----|
| CA | 10 |
| OR | -10 |
| WA | 2.5 |
| ZZ | 15 |
| ID | |
| | 4 |

**Transformation:**

**Invalid State values**: You can test for invalid values for State using the following:

| Transformation Name | New formula |
|---------------------|-------------|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | ISMISMATCHED (State, 'State') |

The above transform flags rows 4 and 6 as mismatched.

> **NOTE:** A missing value is not valid for a type, including String type.

**Invalid Integer values:** You can test for valid matches for Qty using the following:

| Transformation Name | New formula |
|---------------------|-------------|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | (ISVALID (Qty, 'Integer') && (Qty > 0)) |

| Parameter: New column name | 'valid_Qty' |
|---|---|

The above transform flags as valid all rows where the `Qty` column is a valid integer that is greater than zero.

**Missing values:** The following transform tests for the presence of missing values in either column:

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | (ISMISSING(State) \|\| ISMISSING(Qty)) |
| Parameter: New column name | 'missing_State_Qty' |

After re-organizing the columns using the `move` transform, the dataset should now look like the following:

| State | Qty | mismatched_State | valid_Qty | missing_State_Qty |
|---|---|---|---|---|
| CA | 10 | false | true | false |
| OR | -10 | false | false | false |
| WA | 2.5 | false | false | false |
| ZZ | 15 | true | true | false |
| ID | | false | false | true |
| | 4 | false | true | true |

Since the data does not contain null values, the following transform generates null values based on the preceding criteria:

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | ((mismatched_State == 'true') \|\| (valid_Qty == 'false') \|\| (missing_State_Qty == 'true')) ? NULL() : 'ok' |
| Parameter: New column name | 'status' |

You can then use the `ISNULL` check to remove the rows that fail the above test:

| Transformation Name | Filter rows |
|---|---|
| Parameter: Condition | Custom formula |
| Parameter: Type of formula | Custom single |
| Parameter: Condition | ISNULL('status') |
| Parameter: Action | Delete matching rows |

**Results:**

Based on the above tests, the output dataset contains one row:

| State | Qty | mismatched_State | valid_Qty | missing_State_Qty | status |
|-------|-----|------------------|-----------|-------------------|--------|
| CA    | 10  | false            | true      | false             | ok     |

# EXAMPLE - Type Parsing Functions

This example shows how to use parsing functions for evaluating input values against the function-specific data type.

**Functions**:

| Item | Description |
|------|-------------|
| PARSEBOOL Function | Evaluates a String input against the Boolean datatype. If the input matches, the function outputs a Boolean value. Input can be a literal, a column of values, or a function returning String values. |
| PARSEDATE Function | Evaluates an input against the default input formats or (if specified) an array of Datetime format strings in their listed order. If the input matches one of the formats, the function outputs a Datetime value. |
| PARSEFLOAT Function | Evaluates a String input against the Decimal datatype. If the input matches, the function outputs a Decimal value. Input can be a literal, a column of values, or a function returning String values. |
| PARSEINT Function | Evaluates a String input against the Integer datatype. If the input matches, the function outputs an Integer value. Input can be a literal, a column of values, or a function returning String values. |

**Source:**

The following table contains data on a series of races.

| raceId | disqualified | date | racerId | time_sc |
|--------|--------------|------|---------|---------|
| 1 | FALSE | 2/1/20 | 1 | 24.22 |
| 2 | f | 2/8/20 | 1 | 25 |
| 3 | no | 2/8/20 | 1 | 24.11 |
| 4 | n | 1-Feb-20 | 2 | 26.1 |
| 5 | TRUE | 8-Feb-20 | 2.2 | -25.22 |
| 6 | t | 2/8/2020  10:16:00 AM | 2 | 25.44 |
| 7 | yes | 2/1/20 | 3 | 24 |
| 8 | y | 2/8/20 | 33 | 29.22 |
| 9 | 0 | 2/8/20 | 3 | 24.78 |
| 10 | 1 | 1-Feb-20 | 4 | 26.2.1 |
| 11 | FALSE | 8-Feb-20 | | 28.22 sec |
| 12 | FALSE | 2/8/2020  10:16:00 AM | 4 | 27.11 |

As you can see, this dataset has variation in values (FALSE, f, no, n) and problems with the data.

**Transformation:**

When the data is first imported, it may be properly typed for each column. To use the parsing functions, these columns should be converted to String data type:

| | |
|------|------|
| **Transformation Name** | Change column data type |
| **Parameter: Columns** | disqualified,date,racerId,time_sc |
| **Parameter: New type** | String |

Now, you can parse individual columns.

disqualified column:

| Transformation Name | Edit column with formula |
|---|---|
| Parameter: Columns | disqualified |
| Parameter: Formula | PARSEBOOL($col) |

racerId column:

| Transformation Name | Edit column with formula |
|---|---|
| Parameter: Columns | racerId |
| Parameter: Formula | PARSEINT($col) |

time_sc column:

| Transformation Name | Edit column with formula |
|---|---|
| Parameter: Columns | time_sc |
| Parameter: Formula | PARSEFLOAT($col) |

date column:

For the date column, the PARSEDATE function supports a default set of Datetime formats. Since some of the listed formats are different from these defaults, you must specify all of the formats. These formats are specified as an array of string values as the second argument of the function:

> **Tip:** For the PARSEDATE function, it's useful to use the Preview to verify that all of the dates in the column are represented in the array of output formats. You can see the available output formats through the data type menu at the top of a column in the Transformer Page.

| Transformation Name | Edit column with formula |
|---|---|
| Parameter: Columns | date |
| Parameter: Formula | PARSEDATE($col, ['yyyy-MM-dd','yyyy\/MM\/dd','M\/d\/yyy hh: mm','MMMM d, yyyy','MMM d, yyyy']) |

After all of the date values have been standardized to the output format of the PARSEDATE function, you may choose to remove the time element of the values:

| Transformation Name | Replace text or pattern |
|---|---|
| Parameter: Column | date |
| Parameter: Find | `` ` {digit}{2}:{digit}{2}:{digit}{2}{end}` `` |
| Parameter: Replace with | '' |

**Results:**

After executing the above steps, the data appears as follows. Notes on each column's output are below the table.

| raceId | disqualified | date | racerId | time_sc |
|---|---|---|---|---|

| 1 | false | 2020-02-01 | 1 | 24.22 |
|---|---|---|---|---|
| 2 | false | 2020-02-08 | 1 | 25 |
| 3 | false | 2020-02-08 | 1 | 24.11 |
| 4 | false | 2020-02-01 | 2 | 26.1 |
| 5 | true | 2020-02-08 | *null* | -25.22 |
| 6 | true | 2020-02-08 | 2 | 25.44 |
| 7 | true | 2020-02-01 | 3 | 24 |
| 8 | true | 2020-02-08 | 33 | 29.22 |
| 9 | false | 2020-02-08 | 3 | 24.78 |
| 10 | true | 2020-02-01 | 4 | *null* |
| 11 | false | 2020-02-08 | *null* | *null* |
| 12 | false | 2020-02-08 | 4 | 27.11 |

disqualified column:

- The PARSEBOOL function normalizes all valid Boolean values to either `false` or `true`.

racerId column:

- The PARSEINT function writes invalid values as null values.
- The function writes empty values as null values.
- The value `33` remains, since it is a valid Integer. This value should be fixed manually.

time_sc:

- The PARSEFLOAT function writes the source value `25.00` as `25` in output.
- The source value `-25.22` remains. However, since this is time-based data, it needs to be fixed.
- Invalid values are written as nulls.

date column:

- All values are written in the standardized format: `yyyy-MM-dd HH:mm:ss`. Time data has been stripped.

# EXAMPLE - UNICODE Function

In this example, you can see how the CHAR function can be used to convert numeric index values to Unicode characters, and the UNICODE function can be used to convert characters back to numeric values.

**Functions:**

| Item | Description |
|------|-------------|
| CHAR Function | Generates the Unicode character corresponding to an inputted Integer value. |
| UNICODE Function | Generates the Unicode index value for the first character of the input string. |

**Source:**

The following column contains some source index values:

| index |
|-------|
| 1 |
| 33 |
| 33.5 |
| 34 |
| 48 |
| 57 |
| 65 |
| 90 |
| 97 |
| 121 |
| 254 |
| 255 |
| 256 |
| 257 |
| 9998 |
| 9999 |

**Transformation:**

When the above values are imported to the Transformer page, the column is typed as integer, with a single mismatched value (33.5). To see the corresponding Unicode characters for these characters, enter the following transformation:

| Transformation Name | New formula |
|---------------------|-------------|
| **Parameter: Formula type** | Single row formula |
| **Parameter: Formula** | CHAR(index) |
| **Parameter: New column** | 'char_index' |

To see how these characters map back to the index values, now add the following transformation:

| Transformation Name | New formula |
|---|---|
| Parameter: Formula type | Single row formula |
| Parameter: Formula | UNICODE(char_index) |
| Parameter: New column name | 'unicode_char_index' |

**Results:**

| index | char_index | unicode_char_index |
|---|---|---|
| 1 | | 1 |
| 33 | ! | 33 |
| 33.5 | | |
| 34 | " | 34 |
| 48 | 0 | 48 |
| 57 | 9 | 57 |
| 65 | A | 65 |
| 90 | Z | 90 |
| 97 | a | 97 |
| 122 | z | 122 |
| 254 | þ | 254 |
| 255 | ÿ | 255 |
| 256 | | 256 |
| 257 | | 257 |
| 9998 | | 9998 |
| 9999 | | 9999 |

Note that the floating point input value was not processed.

# EXAMPLE - Unixtime Functions

This example illustrates how you can use functions to manipulate Unix time values in a column of Datetime type.

**Functions**:

| Item | Description |
|------|-------------|
| UNIXTIME Function | Derives the Unixtime (or epoch time) value from a Datetime value. Source value can be a reference to a column containing Datetime values. |

**Source:**

| date |
|------|
| 2/8/16 15:41 |
| 12/30/15 0:00 |
| 4/26/15 7:07 |

**Transformation:**

Use the following transformation step to generate a column containing the above values as Unix timecode values:

| Transformation Name | New formula |
|---------------------|-------------|
| **Parameter: Formula type** | Single row formula |
| **Parameter: Formula** | UNIXTIME (date) |
| **Parameter: New column name** | 'unixtime_date' |

**Results:**

> **NOTE:** If the source Datetime value does not contain a valid input for one of these functions, no value is returned.

| date | unixtime_date |
|------|---------------|
| 2/8/16 15:41 | 1454946120000 |
| 12/30/15 0:00 | 1451433600000 |
| 4/26/15 7:07 | 1430032020000 |

# EXAMPLE - Unnest an Array

This section describes how to unnest the values in an Array into separate columns in your dataset.

**Source:**

In the following example dataset, students took the same test three times, and their scores were stored in any array in the `Scores` column.

| LastName | FirstName | Scores |
|---|---|---|
| Adams | Allen | [81,87,83,79] |
| Burns | Bonnie | [98,94,92,85] |
| Cannon | Chris | [88,81,85,78] |

**Transformation:**

When the data is imported, you might have to re-type the `Scores` column as an array:

| Transformation Name | Change column data type |
|---|---|
| Parameter: Columns | Scores |
| Parameter: New type | Array |

You can now unnest the `Scores` column data into separate columns:

| Transformation Name | Unnest Objects into columns |
|---|---|
| Parameter: Column | Scores |
| Parameter: Parameter: Paths to elements | [0] |
| Parameter: Parameter: Paths to elements | [1] |
| Parameter: Parameter: Paths to elements | [2] |
| Parameter: Parameter: Paths to elements | [3] |
| Parameter: Remove elements from original | true |
| Parameter: Include original column name | true |

In the above transformation:

- Each path is specified in a separate row.
  - The `[x]` syntax indicates that the path is the *xth* element of the array.
  - The first element of an array is referenced using `[0]`.
- You can choose to delete the element from the original or not. Deleting the element can be a helpful way of debugging your transformation. If all of the elements are gone, then the transformation is complete.
- If you include the original column name in the output column names, you have some contextual information for the outputs.

**Results:**

| LastName | FirstName | Scores_0 | Scores_1 | Scores_2 | Scores_3 |
|----------|-----------|----------|----------|----------|----------|
| Adams | Allen | 81 | 87 | 83 | 79 |
| Burns | Bonnie | 98 | 94 | 92 | 85 |
| Cannon | Chris | 88 | 81 | 85 | 78 |

# EXAMPLE - Unnest JSON Records

You can unnest a set of JSON records into new columns of tabular data for easier manipulation within the application.

The following example contains records of messages about individual diet and exercise achievements:

```
{
  "object": [
    {
      "score": 19669,
      "title": "M/07/1'3\" [23lbs > 13lbs = 10lbs] Still a bit to go, but my owner no longer refers to me as
his chunky boy!",
      "ups": 19669,
      "id": "9kt8ex",
      "url": "https://i.redd.it/bzygw285fpp11.jpg",
      "short": "bzygw285fpp11.jpg"
    },
    {
      "score": 19171,
      "title": "M/29/5'11\" [605 pounds > 375 pounds = 230 pounds lost] (14 months) Still considered super
morbidly obese but I've made some good progress.",
      "ups": 19171,
      "id": "9x2774",
      "url": "https://i.redd.it/wbbufmll0cy11.jpg",
      "short": "wbbufmll0cy11.jpg"
    },
    {
      "score": 16778,
      "title": "F/28/5\u20197\u201d [233lbs to 130lbs] Got tired of being obese and took control of my life!",
      "ups": 16778,
      "id": "a8guou",
      "url": "https://i.redd.it/3t0kmljnmq521.jpg",
      "short": "3t0kmljnmq521.jpg"
    },
    {
      "score": 16743,
      "title": "M/22/5'11\" [99lbs > 150lbs = 51lbs] Anorexia my recovery",
      "ups": 16743,
      "id": "atla3n",
      "url": "https://i.redd.it/9t6tvsjs16i21.jpg",
      "short": "9t6tvsjs16i21.jpg"
    }
  ]
}
```

The outer JSON is a single key-value pair:

- key: `object`
- value: array of JSON records

When source JSON records structured in this manner are imported, each JSON record in the object is imported into a separate row.  You can unnest this data by applying an Unnest values transformation.

> **NOTE**: The object can contain only one nested array of JSON data. If the object contains multiple nested arrays, it is not not broken into separate rows. All unnesting must be performed in your recipe steps

Suppose you want to compute the average of all workout scores. First, you must unnest the JSON records and then apply the `AVERAGE` function.

**Steps:**

> **Tip:** The easiest way to unnest is to select the column header for the column containing your data. After you select the column header, you are provided with suggestions to Unnest Values into new columns. You can use the Unnest suggestion and click **Add**. The following steps illustrate how to create this transformation manually.

1. In the Recipe panel, click **New Step**.
2. In the Search panel, enter `unnest values into new columns`.
3. Specify the following transformation. Substitute the Paths to elements values below with the top-level keys in your JSON records:

| Transformation Name | `Unnest values into new columns` |
|---|---|
| **Parameter: Column** | `object` |
| **Parameter: Path to elements** | `id` |
| **Parameter: Path to elements** | `score` |
| **Parameter: Path to elements** | `short` |
| **Parameter: Path to elements** | `title` |
| **Parameter: Paths to elements** | `ups` |
| **Parameter: Path to elements** | `url` |

4. The above step breaks out the key-value pairs for the specified keys into separate columns in the dataset. Each Paths to elements entry specifies a key in the JSON record, which is used to create a new column of the same name. The key's associated value becomes a cell value in the new column.
5. You can now delete the source column. In the example, the source column is `object`.

> **Tip:** You can choose to remove the original from the source or not. In deeper or wider JSON files, removing can help to identify what remains to be unnested. When you're done unnesting a column and have removed data from the original, you should have an empty column.

Your table should look like the following:

| id | score | short | title | ups | url |
|---|---|---|---|---|---|
| 9kt 8ex | 19669 | bzygw285 fpp11.jpg | M/07/1'3" [23lbs > 13lbs = 10lbs] Still a bit to go, but my owner no longer refers to me as his chunky boy! | 19669 | *https://i.redd.it /bzygw285fpp1 1.jpg* |
| 9x2 774 | 19171 | wbbufmll0 cy11.jpg | M/29/5'11" [605 pounds > 375 pounds = 230 pounds lost] (14 months) Still considered super morbidly obese but I've made some good progress. | 19171 | *https://i.redd.it /wbbufmll0cy11 .jpg* |
| a8g uou | 16778 | 3t0kmljnm q521.jpg | F/28/5'7" [233lbs to 130lbs] Got tired of being obese and took control of my life! | 16778 | *https://i.redd.it /3t0kmljnmq52 1.jpg* |
| atla 3n | 16743 | 9t6tvsjs16 i21.jpg | M/22/5'11" [99lbs > 150lbs = 51lbs] Anorexia my recovery | 16743 | *https://i.redd.it /9t6tvsjs16i21. jpg* |

Now, you can apply the average function:

| Transformation Name | `New formula` |
|---|---|
| **Parameter: Formula type** | `Single row formula` |
| **Parameter: Formula** | `AVERAGE(score)` |

| | Parameter: New column name | Average_score |
|---|---|---|

**Results**:

| id | score | short | title | ups | url | Average_score |
|---|---|---|---|---|---|---|
| 9kt 8ex | 19669 | bzygw28 5fpp11. jpg | M/07/1'3" [23lbs > 13lbs = 10lbs] Still a bit to go, but my owner no longer refers to me as his chunky boy! | 19669 | *https://i.redd. it /bzygw285fp p11.jpg* | 18090.25 |
| 9x2 774 | 19171 | wbbufmll 0cy11. jpg | M/29/5'11" [605 pounds > 375 pounds = 230 pounds lost] (14 months) Still considered super morbidly obese but I've made some good progress. | 19171 | *https://i.redd. it /wbbufmll0cy 11.jpg* | 18090.25 |
| a8g uou | 16778 | 3t0kmljn mq521. jpg | F/28/5'7" [233lbs to 130lbs] Got tired of being obese and took control of my life! | 16778 | *https://i.redd. it /3t0kmljnmq5 21.jpg* | 18090.25 |
| atla 3n | 16743 | 9t6tvsjs1 6i21.jpg | M/22/5'11" [99lbs > 150lbs = 51lbs] Anorexia my recovery | 16743 | *https://i.redd. it /9t6tvsjs16i2 1.jpg* | 18090.25 |

If needed, these records can be re-nested into JSON records for downstream consumption.

# EXAMPLE - Extract Keys From Objects

You can extract the keys from an Object column into an Array of String values.

**Functions:**

| Item | Description |
|------|-------------|
| KEYS Function | Extracts the key values from an Object data type column and stores them in an array of String values. |

**Source:**

The following dataset contains configuration blocks for individual features, each of which has a different configuration. These example blocks are of Object type.

> **Tip:** In the following example configuration, the **keys** are the values on the left (e.g. `enabled`, `maxRows`, and `maxCols`), while the **values** for those keys are on the right side.

Code formatting has been applied to the Object data to improve legibility.

| FeatureName | Configuration |
|-------------|---------------|
| Whiz Widget | ``` { "enabled": "true", "maxRows": "1000", "maxCols": "100" } ``` |
| Magic Button | ``` { "enabled": "false", "startDirectory": "/home", "maxDepth": "15" } ``` |
| Happy Path Finder | ``` { "enabled": "true" } ``` |

**Transformation:**

The following transformation extracts the keys from the Object data in the `Configuration` column.

| Transformation Name | New formula |
|---------------------|-------------|
| **Parameter: Formula type** | Single row formula |
| **Parameter: Formula** | keys(Configuration) |
| **Parameter: New column name** | 'keys_Configuration' |

**Results:**

The `keys_Configuration` column contains the arrays of the key values.

| FeatureName | Configuration | keys_Configuration |
|---|---|---|
| Whiz Widget | ```<br>{<br> "enabled": "true",<br> "maxRows": "1000",<br> "maxCols": "100"<br>}<br>``` | `["enabled","maxRows","maxCols"]` |
| Magic Button | ```<br>{<br> "enabled": "false",<br> "startDirectory": "/home",<br> "maxDepth": "15"<br>}<br>``` | `["enabled","startDirectory","maxDepth"]` |
| Happy Path Finder | ```<br>{<br> "enabled": "true"<br>}<br>``` | `["enabled"]` |

# EXAMPLE - Extract Object Values

This simple example demonstrates how to extract nested values from Object elements into a separate column.

**Source**:

For example, suppose your restaurant dataset includes a set of characteristics in the `restFeatures` column in the following JSON format, from which you are interested in the total number of seats in the restaurant.

This example contains the data for a single restaurant, formatted as regular JSON, for simplicity:

```
{
  "Credit":"Y",
  "Accessible":"Y",
  "Restrooms":"Y",
  "EatIn":"Y",
  "ToGo":"N",
  "AlcoholBeer":"Y",
  "AlcoholHard":"N",
  "TotalTables":"10",
  "TotalTableSeats":"36",
  "Counter":"Y",
  "CounterSeats":"8"
}
```

**Transformation:**

You can use the following transformation to extract the values from `TotalTableSeats` and `CounterSeats` into separate columns:

> **NOTE:** Change the column type to Object before applying the following transformation.

> **NOTE**: Each key must be entered on a separate line in the Path to elements area.

| Transformation Name | Unnest Objects into columns |
|---|---|
| Parameter: Column | restFeatures |
| Parameter: Paths to elements | TotalTableSeats |
| Parameter: Paths to elements | CounterSeats |
| Parameter: Include original column name | Selected |

**Results**:

| restFeatures_TotalTable Seats | restFeatures_CounterSeats |
|---|---|
| 36 | 8 |

After converting into separate columns, you can perform a simple sum of the `TotalTableSeats` and `CounterSeats` columns to determine the total number of seats in the restaurant.

The final table looks like the following:

| restFeatures_TotalTable Seats | restFeatures_CounterSeats | TotalSeats_Restaurant |
|---|---|---|
|  |  |  |

| 36 | 8 | 44 |

# EXAMPLE - Filtering Strings into Objects

You can create nested objects by filtering strings. In this example, column headers and column values are nested into a single entity in a new column of Object data type.

**Functions:**

| Item | Description |
|---|---|
| FILTEROBJECT Function | Filters the keys and values from an Object data type column based on a specified key value. |
| PARSEOBJECT Function | Evaluates a String input against the Object datatype. If the input matches, the function outputs an Object value. Input can be a literal, a column of values, or a function returning String values. |

**Source:**

The following table shows a series of requests for inventory on three separate products. These are rolling requests, so inventory levels in the subsequent request are decreased based on the previous request.

| date | reqProdId | reqValue | prodA | prodB | prodC |
|---|---|---|---|---|---|
| 5/10/21 | prodA | 10 | 90 | 100 | 100 |
| 5/10/21 | prodC | 20 | 90 | 100 | 80 |
| 5/10/21 | prodA | 15 | 75 | 100 | 80 |
| 5/11/21 | prodB | 25 | 75 | 75 | 80 |
| 5/11/21 | prodA | 5 | 70 | 75 | 80 |
| 5/11/21 | prodC | 30 | 70 | 75 | 50 |
| 5/12/21 | prodB | 10 | 70 | 65 | 50 |

You must create a column containing the request information and the inventory level information for the requested product after the request has been fulfilled.

**Transformation:**

The five data columns must be nested into an Object. The generated column is called `inventoryLevels`.

| | |
|---|---|
| **Transformation Name** | `Nest columns into Objects` |
| **Parameter: Columns** | `reqProdId,reqValue,prodA,prodB,prodC` |
| **Parameter: Nest columns to** | `Object` |
| **Parameter: New column name** | `inventoryLevels` |

You can then build the inventory response column (`inventoryResponse`) using the FILTEROBJECT function:

| | |
|---|---|
| **Transformation Name** | `New formula` |
| **Parameter: Formula type** | `Single row formula` |
| **Parameter: Formula** | `filterobject(parseobject(inventoryRequest),['reqProdId','reqValue',reqProdId])` |
| **Parameter: New column name** | `inventoryResponse` |

**Results:**

The `inventoryResponse` column contains the request information and the response information after the request has been fulfilled.

| date | reqProdId | reqValue | prodA | prodB | prodC | inventoryLevels | inventoryResponse |
|------|-----------|----------|-------|-------|-------|-----------------|-------------------|
| 5/10 /21 | prodA | 10 | 90 | 100 | 100 | {"reqProdId":"prodA","reqValue":"10","prodA":"90","prodB":"100","prodC":"100"} | {"reqProdId":"prodA","reqValue":"10","prodA":"90"} |
| 5/10 /21 | prodC | 20 | 90 | 100 | 80 | {"reqProdId":"prodC","reqValue":"20","prodA":"90","prodB":"100","prodC":"80"} | {"reqProdId":"prodC","reqValue":"20","prodC":"80"} |
| 5/10 /21 | prodA | 15 | 75 | 100 | 80 | {"reqProdId":"prodA","reqValue":"15","prodA":"75","prodB":"100","prodC":"80"} | {"reqProdId":"prodA","reqValue":"15","prodA":"75"} |
| 5/11 /21 | prodB | 25 | 75 | 75 | 80 | {"reqProdId":"prodB","reqValue":"25","prodA":"75","prodB":"75","prodC":"80"} | {"reqProdId":"prodB","reqValue":"25","prodB":"75"} |
| 5/11 /21 | prodA | 5 | 70 | 75 | 80 | {"reqProdId":"prodA","reqValue":"5","prodA":"70","prodB":"75","prodC":"80"} | {"reqProdId":"prodA","reqValue":"5","prodA":"70"} |
| 5/11 /21 | prodC | 30 | 70 | 75 | 50 | {"reqProdId":"prodC","reqValue":"30","prodA":"70","prodB":"75","prodC":"50"} | {"reqProdId":"prodC","reqValue":"30","prodC":"50"} |
| 5/12 /21 | prodB | 10 | 70 | 65 | 50 | {"reqProdId":"prodB","reqValue":"10","prodA":"70","prodB":"65","prodC":"50"} | {"reqProdId":"prodB","reqValue":"10","prodB":"65"} |

alteryx | TRIFACTA