Informatica® Cloud Data Integration
November 2024

# Function Reference

Informatica Cloud Data Integration Function Reference
November 2024
April 2024



Publication Date: 2024-11-15

# Table of Contents

# Preface

Refer to the *Function Reference* for information about the transformation language used to write functions to transform data with Data Integration. Learn how to use functions, operators, and constants to write simple or complex transformation expressions.

# Informatica Resources

Informatica provides you with a range of product resources through the Informatica Network and other online portals. Use the resources to get the most from your Informatica products and solutions and to learn from other Informatica users and subject matter experts.

## Informatica Documentation

Use the Informatica Documentation Portal to explore an extensive library of documentation for current and recent product releases. To explore the Documentation Portal, visit https://docs.informatica.com.

If you have questions, comments, or ideas about the product documentation, contact the Informatica Documentation team at infa_documentation@informatica.com.

## Informatica Intelligent Cloud Services web site

You can access the Informatica Intelligent Cloud Services web site at http://www.informatica.com/cloud. This site contains information about Informatica Cloud integration services.

## Informatica Intelligent Cloud Services Communities

Use the Informatica Intelligent Cloud Services Community to discuss and resolve technical issues. You can also find technical tips, documentation updates, and answers to frequently asked questions.

Access the Informatica Intelligent Cloud Services Community at:

https://network.informatica.com/community/informatica-network/products/cloud-integration

Developers can learn more and share tips at the Cloud Developer community:

https://network.informatica.com/community/informatica-network/products/cloud-integration/cloud-developers

## Informatica Intelligent Cloud Services Marketplace

Visit the Informatica Marketplace to try and buy Data Integration Connectors, templates, and mapplets:

https://marketplace.informatica.com/

## Data Integration connector documentation

You can access documentation for Data Integration Connectors at the Documentation Portal. To explore the Documentation Portal, visit https://docs.informatica.com.

## Informatica Knowledge Base

Use the Informatica Knowledge Base to find product resources such as how-to articles, best practices, video tutorials, and answers to frequently asked questions.

To search the Knowledge Base, visit https://search.informatica.com. If you have questions, comments, or ideas about the Knowledge Base, contact the Informatica Knowledge Base team at KB_Feedback@informatica.com.

## Informatica Intelligent Cloud Services Trust Center

The Informatica Intelligent Cloud Services Trust Center provides information about Informatica security policies and real-time system availability.

You can access the trust center at https://www.informatica.com/trust-center.html.

Subscribe to the Informatica Intelligent Cloud Services Trust Center to receive upgrade, maintenance, and incident notifications. The Informatica Intelligent Cloud Services Status page displays the production status of all the Informatica cloud products. All maintenance updates are posted to this page, and during an outage, it will have the most current information. To ensure you are notified of updates and outages, you can subscribe to receive updates for a single component or all Informatica Intelligent Cloud Services components. Subscribing to all components is the best way to be certain you never miss an update.

To subscribe, on the Informatica Intelligent Cloud Services Status page, click **SUBSCRIBE TO UPDATES**. You can choose to receive notifications sent as emails, SMS text messages, webhooks, RSS feeds, or any combination of the four.

## Informatica Global Customer Support

You can contact a Global Support Center through the Informatica Network or by telephone.

To find online support resources on the Informatica Network, click **Contact Support** in the Informatica Intelligent Cloud Services Help menu to go to the **Cloud Support** page. The **Cloud Support** page includes system status information and community discussions. Log in to Informatica Network and click **Need Help** to find additional resources and to contact Informatica Global Customer Support through email.

The telephone numbers for Informatica Global Customer Support are available from the Informatica web site at https://www.informatica.com/services-and-training/support-services/contact-us.html.

CHAPTER 1

# Function reference

Informatica provides a transformation language that includes SQL-like functions to transform source data. Use these functions to write expressions and create user-defined functions.

Expressions modify data or test whether data matches conditions. For example, you can use the AVG function to calculate the average salary of all the employees or the SUM function to calculate the total sales for a specific branch. You can create a simple expression that only contains a single function, such as AVG. You can also write complex expressions by nesting functions within functions.

**Note:** If you create expressions within a mapping in SQL ELT mode, use your cloud data warehouse's native functions and expression syntax instead of Informatica functions and expression syntax. For more information about the native functions and expression syntax, see the documentation for your cloud data warehouse.

User-defined functions use expression logic to build complex expressions. You can include them in other user-defined functions or in expressions. For more information about user-defined functions, see *Components*.

## Function components

The transformation language includes the following components to create simple or complex transformation expressions:

- **Functions.**Over 100 SQL-like functions allow you to change data in a mapping.
- **Operators.**Use transformation operators to create transformation expressions to perform mathematical computations, combine data, or compare data.
- **Constants.**Use built-in constants to reference values that remain constant, such as TRUE.
- **System Variables**. Use variables to return system values that change, such as the current task name and run ID.

CHAPTER 2

# Constants

This chapter includes the following topics:



## FALSE

Clarifies a conditional expression. FALSE is equivalent to the integer 0.

### Example

The following example uses FALSE in a DECODE expression to verify that values are between 25 and 30. This is useful if you want to perform multiple searches based on a single search value:

```
DECODE (FALSE,
Var1 >= 25, 'Var1 not in desired range.',
Var1 <= 30, 'Var1 not in desired range.',
'Var1 in desired range.')
```

## NULL

Indicates that a value is either unknown or undefined. NULL is not equivalent to a blank or empty string (for character columns) or 0 (for numerical columns).

Although you can write expressions that return nulls, any column that has the NOT NULL or PRIMARY KEY constraint will not accept nulls. Therefore, if Data Integration tries to write a null value to a column with one of these constraints, the database will reject the row and Data Integration will write it to the reject file. Be sure to consider nulls when you create transformations.

Functions can handle nulls differently. If you pass a null value to a function, it might return 0 or NULL, or it might ignore null values.

## Working with null values in Boolean expressions

Expressions that combine a null value with a Boolean expression produces results that are ANSI compliant. For example:

- NULL AND TRUE = NULL
- NULL AND FALSE = FALSE

## Null values in comparison expressions

When you use a null value in an expression that contains a comparison operator, Data Integration produces a null value. However, you can also configure the Data Integration Server service custom property Treat Null in Comparison Operators As to tell Data Integration to treat null values as high or low in comparison operations.

Use the Treat Null in Comparison Operators As custom DTM property to configure how Data Integration handles null values in comparison expressions. This property affects the behavior of the following comparison operators in expressions:

```
=, !=, ^=, <>, >, >=, <, <=
```

For example, consider the following expressions:

```
NULL > 1
NULL = NULL
```

The following table describes how Data Integration evaluates the expressions:

| Expression | Treat Null in Comparison Operators As | | |
|---|---|---|---|
| | NULL | HIGH | LOW |
| NULL > 1 | NULL | TRUE | FALSE |
| NULL = NULL | NULL | TRUE | TRUE |

## Null values in filter conditions

If a filter condition evaluates to NULL, the function does not select the record.

## Nulls with operators

Any expression that uses operators (except the string operator ||) and contains a null value always evaluates to NULL. For example, the following expression evaluates to NULL:

```
8 * 10 - NULL
```

To test for nulls, use the ISNULL function.

# TRUE

Returns a value based on the result of a comparison. TRUE is equivalent to the integer 1.

## Example

The following example uses TRUE in a DECODE expression to return values based on the results of a comparison. This is useful if you want to perform multiple searches based on a single search value:

```
DECODE( TRUE,
Var1 = 22,'Variable 1 was 22!',
Var2 = 49,'Variable 2 was 49!',
Var1 < 23, 'Variable 1 was less than 23.',
Var2 > 30, 'Variable 2 was more than 30.',
'Variables were out of desired ranges.')
```

CHAPTER 3

# Operators

This chapter includes the following topics:



## Operator precedence

When you create an expression, you can use multiple operators and use operators within nested expressions.

If you write an expression that includes multiple operators, Data Integration evaluates the expression in the following order:

1. Complex operators
2. Arithmetic operators
3. String operators
4. Comparison operators
5. Logical operators

Data Integration evaluates operators in the order they appear in the following table. It evaluates operators in an expression with equal precedence to all operators from left to right.

The following table lists the precedence for all transformation language operators:

| Operator | Meaning |
|---|---|
| [ ], . | Subscript, dot. |
| ( ) | Parentheses. |
| +, -, NOT | Unary plus and minus and the logical NOT operator. |
| *, /, % | Multiplication, division, modulus. |

| Operator | Meaning |
|---|---|
| +, - | Addition, subtraction. |
| \|\| | Concatenate. |
| <, <=, >, >= | Less than, less than or equal to, greater than, greater than or equal to. |
| =, <>, !=, ^= | Equal to, not equal to, not equal to, not equal to. |
| AND | Logical AND operator, used when specifying conditions. |
| OR | Logical OR operator, used when specifying conditions. |

You can use operators within nested expressions. When expressions contain parentheses, Data Integration evaluates operations inside parentheses before operations outside parentheses. Operations in the innermost parentheses are evaluated first.

For example, depending on how you nest the operations, the equation 8 + 5 - 2 * 8 returns different values:

| Equation | Return value |
|---|---|
| 8 + 5 - 2 * 8 | -3 |
| 8 + (5 - 2) * 8 | 32 |

# Complex operators

Use complex operators to access elements in an array, map, or struct. You can use complex operators only in advanced mode.

The following table lists the complex operators:

| Operator | Meaning |
|---|---|
| [ ] | Subscript operator.<br>Use a subscript operator with an array or a map:<br>- In an array, use a subscript operator to access one or more elements.<br>- In a map, use a subscript operator to access the value corresponding to a given key in a key-value pair. |
| . | Dot operator.<br>Use a dot operator with a struct or an array of structs:<br>- In a struct, use a dot operator to access an element.<br>- In an array of structs, use a dot operator to access elements in each struct. The operator returns the elements of the same name within each struct as an array. |

To access elements in a nested hierarchy, you can use a combination of complex operators.

# Subscript operator (Arrays)

Use a subscript operator to access elements in an array. You can access a specific element or a range of elements.

## Syntax

To access a specific element in an array, use the following syntax:

```
array[ index ]
```

To access a range of elements in an array, use the following syntax:

```
array[ start_index , end_index ]
```

The following table describes the arguments in the syntax:

| Argument | Description |
|---|---|
| array | Array data type. The array from which you want to access one or more elements.<br>You can enter any valid expression that evaluates to an array. |
| index | Integer data type. The position of the element that you want to access. For example, an index of 0 indicates the first element in an array. |
| start_index | Integer data type. The starting index in a range of elements that you want to access. The subscript operator includes the element that the starting index represents. |
| end_index | Integer data type. The ending index in a range of elements that you want to access. The subscript operator excludes the element that the ending index represents. |

You can use an expression for the index that returns an integer value. If the expression returns a negative value, the index is considered to be 0.

If the specified index is greater than the size of the array minus 1, the index accesses the final element in the array.

## Return Value

Element in the array. The return type is the same as the data type of the element.

If you specify two indices separated by a comma, such as `[i,j]`, the expression returns an array of the elements from `i` to `j-1`. If `i` is greater than `j` or the size of the array, the expression returns an empty array.

NULL in the following situations:

- The index is greater than the size of the array.
- The index is NULL.
- You specify multiple indices such as `[i,j]` and either `i` or `j` is NULL.
- The array is NULL.

## Example

You have the following array of strings:

```
drinks = ['milk', 'coffee', 'tea', 'chai']
```

The following expressions use a subscript operator to access string elements in the array:

| Input Value | RETURN VALUE |
|---|---|
| drinks[0] | 'milk' |
| drinks[2] | 'tea' |
| drinks[NULL] | NULL |
| drinks[1,3] | ['coffee','tea'] |
| drinks[2,NULL] | NULL |
| drinks[3,1] | [ ] |

# Subscript operator (Maps)

Use a subscript operator to access the value corresponding to a given key in a key-value pair.

## Syntax

To access the value corresponding to a given key in a map, use the following syntax:

```
map[ key ]
```

The following table describes the arguments in the syntax:

| Argument | Description |
|---|---|
| map | Map data type. The map from which you want to retrieve the value corresponding to a key. |
| key | Data type of the key. The key element for which you want to retrieve the value.<br>You can enter any valid expression that evaluates to a key value of the map data. |

## Return Value

The value associated with the key in the map. The return type is the same as the data type of the value.

NULL if the key does not exist in the map.

## Example

You have the following map:

```
country_currency = [‘England’ -> ‘Pound’, ‘France’ -> ‘Euro’, ‘Japan’ -> ‘Yen’>, ‘USA’ -
> ‘Dollar’]
```

The following expressions use a subscript operator to access values in the map:

| Input Value | RETURN VALUE |
|---|---|
| country_currency [‘Japan’] | ‘Yen’ |
| country_currency [‘India’] | NULL |
| country_currency [‘England’] | ‘Pound’ |

# Dot operator (Structs)

Use a dot operator to access an element in a struct.

## Syntax

To access an element in a struct, use the following syntax:

```
struct.element
```

The following table describes the arguments in the syntax:

| Argument | Description |
|---|---|
| struct | Struct data type. Struct from which you want to access an element.<br>You can enter any valid transformation expression that evaluates to a struct. |
| element | Name of the struct element that you want to access. |

## Return Value

Element in the struct. The return type is the same as the data type of the element.

NULL in the following situations:

- The element in the struct has a NULL value.
- The struct is NULL.

## Examples

You have the following struct:

```
location{
    street: NULL
    city : 'NEWYORK'
    state: 'NY'
    zip : 12345
}
```

The following expressions use a dot operator to access elements in the struct:

| Input Value | RETURN VALUE |
|---|---|
| location.street | NULL |
| location.city | 'NEWYORK' |
| location.state | 'NY' |
| location.zip | 12345 |

# Dot operator (Arrays of structs)

Use a dot operator with an array of structs to access elements from each struct in the array.

## Syntax

To access an element in an array of structs, use the following syntax:

```
array_of_structs.element
```

The following table describes the arguments in the syntax:

| Argument | Description |
|---|---|
| array_of_structs | Array data type. Array of structs from which you want to access elements in each struct. You can enter any valid transformation expression that evaluates to an array. |
| element | Name of the struct element that you want to access. |

### Return Value

Array that contains the specified element from each struct.

NULL in the following situations:

- The element in the struct has a NULL value.
- The struct is NULL.

### Examples

You have the following array with three struct elements and each struct has three elements:

```
employee_info_array = [

 derrick_struct{
     name: 'Derrick'
     city: NULL
     state: 'NY'
 },

 kevin_struct{
     name: 'Kevin'
     city: 'Redwood City'
     state: 'CA'
 },

 lauren_struct{
     name: 'Lauren'
     city: 'Woodcliff Lake'
     state: NULL
 }

]
```

The following expressions use a dot operator to access the elements in each struct in the array:

| Input Value | RETURN VALUE |
|---|---|
| employee_info_array.name | ['Derrick','Kevin','Lauren'] |
| employee_info_array.city | [NULL,'Redwood City','Woodcliff Lake'] |
| employee_info_array.state | ['NY','CA',NULL] |

## Complex operators for nested hierarchies

A nested hierarchy contains elements that also contain hierarchical data, such as an array of structs. Use a combination of complex operators to access elements in a nested hierarchy.

You can access elements in the following types of nested hierarchies:

- Multidimensional array

- Array of structs
- Struct with array elements
- Nested struct

## Multidimensional array

A multidimensional array is an array of arrays. You can use a subscript operator to access a primitive element in an array at the innermost level. You can also use a subscript operator to access an array at any level.

You can use subscript operators to return the following values:

- A primitive element in an array at the innermost level.
- One or more arrays at any level.
- A subset of one or more arrays at any level.

To access a primitive element in an array at the innermost level, you use more than one subscript operator. The number of dimensions in a multidimensional array determines the number of subscript operators to use. Each subscript operator must contain one index value. The data type of the return value is the same as the data type of the primitive elements in the array.

For example, in a two-dimensional array, you use two subscript operators. The first subscript operator accesses the parent array. The second subscript operator accesses the child array within the parent array.

### Examples

Consider the following two-dimensional parent array that contains three child arrays and each child array contains string elements:

```
menu_array = [
 ['milk','coffee','tea','chai'],
 ['ham','turkey',NULL],
 ['caesar','cobb','greek','chipotle']
]
```

You can use subscript operators to access the following types of elements:

**Primitive elements**

The following expressions use two subscript operators to access a specific string element from each child array in the parent array `menu_array`:

| Input Value | RETURN VALUE |
|---|---|
| `menu_array[0][1]` | `'coffee'` |
| `menu_array[2][3]` | `'chipotle'` |
| `menu_array[1][2]` | `NULL` |

#### Array elements

The following expressions use a single subscript operator to access the child arrays in the parent array `menu_array`:

| Input Value | RETURN VALUE |
|---|---|
| `menu_array[0]` | `['milk','coffee','tea','chai']` |
| `menu_array[0,2]` | `[ ['milk','coffee','tea','chai'], ['ham','turkey',NULL] ]` |
| `menu_array[1,0]` | `[ ]` |
| `menu_array[NULL,2]` | `NULL` |

#### Subset of array elements

The following expressions use two subscript operators to access a subset of child arrays in the parent array `menu_array`:

| Input Value | RETURN VALUE |
|---|---|
| `menu_array[0][0,2]` | `['milk','coffee']` |
| `menu_array[2][0,3]` | `['caesar','cobb','greek']` |
| `menu_array[0,2][0,3]` | `[ ['milk','coffee','tea'], ['ham','turkey',NULL] ]` |

## Array of structs

An array of structs is an array with struct elements. Use a combination of subscript and dot operators to access a child struct and the elements in the child struct.

To access an element in a child struct within a parent array, use a subscript operator followed by a dot operator. You can also reverse the order of the operators without changing the return value.

### Examples

You have the following array of structs `employee_info_array`:

```
employee_info_array = [
derrick_struct{
    name: 'Derrick'
    city: NULL
    state: 'NY'
},

kevin_struct{
    name: 'Kevin'
    city: 'Redwood City'
    state: 'CA'
},

lauren_struct{
    name: 'Lauren'
```

```
    city: 'Woodcliff Lake'
    state: NULL
}
]
```

You can access an element in one of the child structs using complex operators in either of the following orders:

**You use a subscript operator and then a dot operator.**

The operators access the array of structs in the following order:

1. The subscript operator accesses the indexed element in the array and returns a struct.
2. The dot operator accesses an element in the struct.

For example, the following expressions use a subscript operator followed by a dot operator to access elements in the array `employee_info_array`:

| Input Value | RETURN VALUE |
|---|---|
| `employee_info_array[0].name` | `'Derrick'` |
| `employee_info_array[1].city` | `'Redwood City'` |
| `employee_info_array[2].state` | `NULL` |

**You use a dot operator and then a subscript operator.**

The operators access the array of structs in the following order:

1. The dot operator locates elements with the same name from each of the structs and returns an array.
2. The subscript operator accesses the indexed element in the array.

For example, the following expressions show the return value when you use a dot operator to access elements in the array `employee_info_array`:

| Input Value | RETURN VALUE |
|---|---|
| `employee_info_array.name` | `['Derrick','Kevin','Lauren']` |
| `employee_info_array.city` | `[NULL,'Redwood City','Woodcliff Lake']` |
| `employee_info_array.state` | `['NY','CA',NULL]` |

The following expressions show the return value when you use a dot operator followed by a subscript operator to access elements in the array `employee_info_array`:

| Input Value | RETURN VALUE |
|---|---|
| `employee_info_array.name[0]` | `'Derrick'` |
| `employee_info_array.city[1]` | `'Redwood City'` |
| `employee_info_array.state[2]` | `NULL` |

Note that the return values are the same whether you use a subscript operator or a dot operator first. For example, the expressions `employee_info_array[0].name` and `employee_info_array.name[0]` have the same return value `'Derrick'`.

## Struct with array elements

To access elements in an array within a struct, use a dot operator followed by a subscript operator. The dot operator first accesses the specified array in the struct. Then, the subscript operator accesses an element in the array based on the index value.

### Example

You have the following struct with arrays `drinks`, `sandwiches`, and `salads`:

```
menu_struct{
drinks: ['milk','coffee','tea','chai']
sandwiches: ['ham','turkey',NULL]
salads: ['caesar','cobb','greek','chipotle']
}
```

If you use the expression `menu_struct.drinks[0]`, the operators access the parent struct and child arrays in the following order:

1. The dot operator accesses the array `drinks`.
2. The subscript operator accesses the value at position 0 in the array `drinks: ['milk','coffee','tea','chai']` and returns `'milk'`.

The following expressions show more examples that use a dot operator followed by a subscript operator to access values in the child arrays in the parent struct `menu_struct`:

| Input Value | RETURN VALUE |
|---|---|
| `menu_struct.drinks[1]` | `'coffee'` |
| `menu_struct.sandwiches[2]` | `NULL` |
| `menu_struct.salads[3]` | `'chipotle'` |
| `menu_struct.drinks[0,3]` | `['milk','coffee','tea']` |

## Nested struct

A nested struct is a struct that contains one or more levels of structs. You can use a dot operator to access a primitive element in a struct at the innermost level. You can also use a dot operator to access a struct at any level.

You can use dot operators to return the following values:

- A primitive element in a struct at the innermost level.
- One or more structs at any level.

To access a primitive element in a struct at the innermost level, you use more than one dot operator. The number of levels in a nested struct determines the number of dot operators to use. The data type of the return value is the same as the data type of the element in the struct.

For example, in a nested struct with two levels of structs, you use two dot operators. The first dot operator accesses the parent struct to locate the child struct. Then, the second dot operator accesses the child struct to return a specific primitive element in the child struct.

### Example

You have the following struct `employee_info_struct` that contains two child structs `home_address_info` and `department_info`:

```
employee_info_struct{

emp_name: 'Derrick'

home_address_info{
    city: 'New York'
    state: NULL

department_info{
    NULL
}

}
```

The following expressions use dot operators to access values from the struct `employee_info_struct`:

| Input Value | RETURN VALUE |
|---|---|
| `employee_info_struct.emp_name` | `'Derrick'` |
| `employee_info_struct.home_address_info` | `{ city: 'New York' state: NULL }` |
| `employee_info_struct.department_info` | `NULL` |
| `employee_info_struct.home_address_info.city` | `'New York'` |
| `employee_info_struct.home_address_info.state` | `NULL` |

# Arithmetic operators

Use arithmetic operators to perform mathematical calculations on numeric data.

The following table lists the transformation language arithmetic operators in order of precedence:

| Operator | Meaning |
|---|---|
| +, - | Unary plus and minus. Unary plus indicates a positive value. Unary minus indicates a negative value. |
| *, /, % | Multiplication, division, modulus. A modulus is the remainder after dividing two integers. For example, 13 % 2 = 1 because 13 divided by 2 equals 6 with a remainder of 1. |
| +, - | Addition, subtraction.<br>The addition operator (+) does not concatenate strings. To concatenate strings, use the string operator \|\|. To perform arithmetic on date values, use the date functions. |

If you perform arithmetic on a null value, the function returns NULL.

When you use arithmetic operators in an expression, all of the operands in the expression must be numeric. For example, the expression `1 + '1'` is not valid because it adds an integer to a string. The expression `1.23 + 4 / 2` is valid because all of the operands are numeric.

# String operators

Use the || string operator to concatenate two strings. The || operator converts operands of any datatype (except Binary) to String datatypes before concatenation:

| Input value | Return value |
|---|---|
| 'alpha' \|\| 'betical' | alphabetical |
| 'alpha' \|\| 2 | alpha2 |
| 'alpha' \|\| NULL | alpha |

The || operator includes leading and trailing spaces. Use the LTRIM and RTRIM functions to trim leading and trailing spaces before concatenating two strings.

## Nulls

The || operator ignores null values. However, if both values are NULL, the || operator returns NULL.

## Example of string operators

The following example shows an expression that concatenates employee first names and employee last names from two columns. This expression removes the spaces from the end of the first name and the beginning of the last name, concatenates a space to the end of each first name, then concatenates the last name:

```
LTRIM( RTRIM( EMP_FIRST ) || ' ' || LTRIM( EMP_LAST ))
```

| EMP_FIRST | EMP_LAST | RETURN VALUE |
|---|---|---|
| ' Alfred' | ' Rice ' | Alfred Rice |
| ' Bernice' | ' Kersins' | Bernice Kersins |
| NULL | ' Proud' | Proud |
| ' Curt' | NULL | Curt |
| NULL | NULL | NULL |

**Note:** You can also use the CONCAT function to concatenate two string values. The || operator, however, produces the same results in less time.

# Comparison operators

Use comparison operators to compare string or numeric strings, manipulate data, and return a TRUE (1) or FALSE (0) value.

The following table lists the transformation language comparison operators:

| Operator | Meaning |
|---|---|
| = | Equal to. |
| > | Greater than. |
| < | Less than. |
| >= | Greater than or equal to. |
| <= | Less than or equal to. |
| <> | Not equal to. |
| != | Not equal to. |
| ^= | Not equal to. |

Use the greater than (>) and less than (<) operators to compare numeric values or return a range of rows based on the sort order for a primary key in a particular field

When you use comparison operators in an expression, the operands must be the same datatype. For example, the expression 123.4 > '123' is not valid because the expression compares a decimal with a string. The expressions 123.4 > 123 and 'a' != 'b' are valid because the operands are the same datatype.

If you compare a value to a null value, the result is NULL.

If a filter condition evaluates to NULL, Data Integration returns NULL.

## Comparing hierarchies

You can use the equal to (=) and not equal to (!=) operators to compare two arrays or two structs.

### Comparing arrays

Two arrays are equivalent if the following conditions are true:

- The array elements are the same data type.
- The arrays are the same size.
- The element at each index is the same.

For example, you have the following arrays:

```
A = [1, 2, 3]
B = [1, 2, 3]
```

You can make the following comparison:

```
A = B
```

Both arrays contain integers, the arrays are the same size, and the element at each index is the same such that `A[0]=B[0], A[1]=B[1], and A[2]=B[2]`. The return value is TRUE (1).

### Comparing structs

Two structs are equivalent if the following conditions are true for the corresponding struct elements:

- The elements are the same data type.
- The elements hold the same data.

**Note:** Two structs are equivalent even if the elements have different names.

For example, you have the following structs:

```
struct1 {
    name:'Paul'
    zip:10004
}

struct2 {
    firstname:'Paul'
    zip1:10004
}
```

You can make the following comparison:

```
struct1 = struct2
```

The corresponding elements are the same data type and hold the same data, so the return value is TRUE (1).

# Logical operators

Use logical operators to manipulate numeric data. Expressions that return a numeric value evaluate to TRUE for values other than 0, FALSE for 0, and NULL for NULL.

The following table lists the transformation language logical operators:

| Operator | Meaning |
|---|---|
| NOT | Negates result of an expression. For example, if an expression evaluates to TRUE, the operator NOT returns FALSE. If an expression evaluates to FALSE, NOT returns TRUE. |
| AND | Joins two conditions and returns TRUE if both conditions evaluate to TRUE. Returns FALSE if one condition is not true. |
| OR | Connects two conditions and returns TRUE if any condition evaluates to TRUE. Returns FALSE if both conditions are not true. |

## Nulls

Expressions that combine a null value with a Boolean expression produce results that are ANSI compliant.

For example, Data Integration produces the following results:

- NULL AND TRUE = NULL
- NULL AND FALSE = FALSE

CHAPTER 4

# Dates

This chapter includes the following topics:



## Dates overview

With the date functions, you can round, truncate, or compare dates, extract one part of a date, or perform arithmetic on a date. You can pass any value with a date datatype to a date function.

Use date variables to capture the current date or session start time on the machine that hosts Data Integration.

The transformation language also provides the following sets of format strings:

**Date format strings**

Use with date functions to specify the parts of a date.

**TO_CHAR format strings**

Use to specify the format of the return string.

**TO_DATE and IS_DATE format strings**

Use to specify the format of a string you want to convert to a date or test.

### Date/Time datatype

The transformation language provides a set of generic datatypes to transform data from different sources. These transformation datatypes include a Date/Time datatype. Data Integration stores dates internally in binary format.

Date functions accept datetime values only. To pass a string to a date function, first use TO_DATE to convert it to a datetime value. For example, the following expression converts a string field to datetime values and then adds one month to each date:

```
ADD_TO_DATE( TO_DATE( STRING_PORT, 'MM/DD/RR'), 'MM', 1 )
```

**Note:** Data Integration supports dates between 1753 A.D. and 9999 A.D.

### Milliseconds

Data Integration supports datetime values up to the second.

If you import a datetime value that includes milliseconds, Data Integration truncates to seconds. If you write a datetime value to a target column that supports milliseconds, Data Integration inserts zeros for the millisecond portion of the date.

## Julian Day, Modified Julian Day, and the Gregorian calendar

Data Integration supports dates in the Gregorian calendar system only. Dates expressed in a different calendar system are not supported.

**Note:** Dates in the Julian calendar are called Julian *dates* and are not supported in Data Integration. This term should not be confused with Julian *Day* or with Modified Julian Day.

The transformation language provides the ability to manipulate Modified Julian Day (MJD) formats using the J format string.

The MJD for a given date is the number of days to that date since Jan 1 4713 BC 00:00:00 (midnight). By definition, MJD includes a time component expressed as a decimal, which represents some fraction of 24 hours. The J format string does not convert this time component.

For example, the following TO_DATE expression converts strings in the SHIP_DATE_MJD_STRING field to date values in the default date format:

```
TO_DATE (SHIP_DATE_MJD_STR, 'J')
```

| SHIP_DATE_MJD_STR | RETURN_VALUE |
|---|---|
| 2451544 | Dec 31 1999 00:00:00 |
| 2415021 | Jan 1 1900 00:00:00 |

Because the J format string does not include the time portion of a date, the return values have the time set to 00:00:00.

You can also use the J format string in TO_CHAR expressions. For example, use the J format string in a TO_CHAR expression to convert date values to MJD values expressed as strings. For example:

```
TO_CHAR(SHIP_DATE, 'J')
```

| SHIP_DATE | RETURN_VALUE |
|---|---|
| Dec 31 1999 23:59:59 | 2451544 |
| Jan 1 1900 01:02:03 | 2415021 |

**Note:** Data Integration ignores the time portion of the date in a TO_CHAR expression.

## Dates in the year 2000

All transformation language date functions support the year 2000. Data Integration supports dates between 1753 A.D. and 9999 A.D.

## RR format string

The transformation language provides the RR format string to convert strings with two-digit years to dates. Using TO_DATE and the RR format string, you can convert a string in the format MM/DD/RR to a date. The RR format string converts data differently depending on the current year.

**Current Year Between 0 and 49**

If the current year is between 0 and 49 (such as 2003) and the source string year is between 0 and 49, Data Integration returns the current century plus the two-digit year from the source string. If the source string year is between 50 and 99, Data Integration returns the previous century plus the two-digit year from the source string.

**Current Year Between 50 and 99**

If the current year is between 50 and 99 (such as 1998) and the source string year is between 0 and 49, Data Integration returns the next century plus the two-digit year from the source string. If the source string year is between 50 and 99, Data Integration returns the current century plus the specified two-digit year.

The following table summarizes how the RR format string converts to dates:

| Current year | Source year | RR format string returns |
|---|---|---|
| 0-49 | 0-49 | Current century |
| 0-49 | 50-99 | Previous century |
| 50-99 | 0-49 | Next century |
| 50-99 | 50-99 | Current century |

## Example of RR

The following expression produces the same return values for any current year between 1950 and 2049:

```
TO_DATE( ORDER_DATE, 'MM/DD/RR' )
```

| ORDER_DATE | RETURN_VALUE |
|---|---|
| '04/12/98' | 04/12/1998 00:00:00 |
| '11/09/01' | 11/09/2001 00:00:00 |

## Difference between the YY and RR format strings

The transformation language also provides a YY format string. Both the RR and YY format strings specify two-digit years. The YY and RR format strings produce identical results when used with all date functions except TO_DATE.

In TO_DATE expressions, RR and YY produce different results.

The following table shows the different results each format string returns:

| String | Current year | TO_DATE(String, 'MM/DD/RR') | TO_DATE(String, 'MM/DD/YY') |
|---|---|---|---|
| 04/12/98 | 1998 | 04/12/1998 00:00:00 | 04/12/1998 00:00:00 |
| 11/09/01 | 1998 | 11/09/2001 00:00:00 | 11/09/1901 00:00:00 |
| 04/12/98 | 2003 | 04/12/1998 00:00:00 | 04/12/2098 00:00:00 |
| 11/09/01 | 2003 | 11/09/2001 00:00:00 | 11/09/2001 00:00:00 |

For dates in the year 2000 and beyond, the YY format string produces less meaningful results than the RR format string. Use the RR format string for dates in the twenty-first century.

## Dates in databases

Although date formats vary from database to database, and even between applications, Data Integration can read any date with a date datatype.

In general, dates stored in databases contain a date and time value. The date includes the month, day, and year, while the time might include the hours, minutes, and seconds. You can pass datetime data to any of the date functions.

## Dates in flat files

The transformation language provides the TO_DATE function to convert strings to datetime values. You can also use IS_DATE to check if a string is a valid date before converting it with TO_DATE.

**Note:** Transformation language date functions accept date values only. If you want to pass a string to a date function, you must first use the TO_DATE function to convert it to a transformation Date/Time datatype.

## Default date format

The application uses a default date format to store and manipulate strings that represent dates. Because Data Integration stores dates in binary format, Data Integration only uses the default date format in certain circumstances.

Data Integration only uses the default date format when you perform the following actions:

**Convert a date to a string by connecting a date/time field to a string field.**

The application converts the date to a string in the default date format, MM/DD/YYYY HH24:MI:SS.

**Convert a string to a date by connecting a string field to a date/time field.**

The application expects the string values to be in the default date format, MM/DD/YYYY HH24:MI:SS. If an input value does not match this format, or it is an invalid date, Data Integration skips the row. If the string is in the default date format, Data Integration converts the string to a date value.

**Use TO_CHAR(date, [format_string]) to convert dates to strings.**

If you omit the format string, Data Integration returns the string in the default date format, MM/DD/YYYY HH24:MI:SS. If you specify a format string, Data Integration returns a string in the specified format.

**Use TO_DATE(date, [format_string]) to convert strings to dates.**

If you omit the format string, Data Integration expects the string in the default date format, MM/DD/YYYY HH24:MI:SS. If you specify a format string, Data Integration expects a string in the specified format.

The default date format of MM/DD/YYYY HH24:MI:SS consists of:

- Month (January = 01, September = 09)
- Day (of the month)
- Year (expressed in four digits, such as 1998)
- Hour (in 24-hour format, for example, 12:00:00AM = 0, 1:00:00AM = 1, 12:00:00PM = 12, 11:00:00PM = 23)
- Minutes
- Seconds

# Date Format Strings

You can evaluate input dates using a combination of format strings and date functions. Date format strings are not internationalized and must be entered in predefined formats as listed in the following table.

The following table summarizes the format strings to specify a part of a date:

| Format String | Description |
|---|---|
| D, DD, DDD, DAY, DY, J | Days (01-31). Use any of these format strings to specify the entire day portion of a date. For example, if you pass 12-APR-1997 to a date function, use any of these format strings specify 12. |
| HH, HH12, HH24 | Hour of day (0-23), where 0 is 12 AM (midnight). Use any of these formats to specify the entire hour portion of a date. For example, if you pass the date 12-APR-1997 2:01:32 PM, use HH, HH12, or HH24 to specify the hour portion of the date. |
| MI | Minutes (0-59). |
| MM, MON, MONTH | Month (01-12). Use any of these format strings to specify the entire month portion of a date. For example, if you pass 12-APR-1997 to a date function, use MM, MON, or MONTH to specify APR. |
| MS | Milliseconds (0-999). |
| NS | Nanoseconds (0-999999999). |
| SS , SSSS | Seconds (0-59). |
| US | Microseconds (0-999999). |
| Y, YY, YYY, YYYY , RR | Year portion of date (1753 to 9999). Use any of these format strings to specify the entire year portion of a date. For example, if you pass 12-APR-1997 to a date function, use Y, YY, YYY, or YYYY to specify 1997. |

**Note:** The format string is not case sensitive. It must always be enclosed within single quotation marks.

The following table describes date functions that use date format strings to evaluate input dates:

| Function | Description |
|---|---|
| ADD_TO_DATE | The part of the date you want to change. |
| DATE_DIFF | The part of the date to use to calculate the difference between two dates. |
| GET_DATE_PART | The part of the date you want to return. This function returns an integer value based on the default date format. |
| IS_DATE | The date you want to check. |
| ROUND (Dates) | The part of the date you want to round. |
| SET_DATE_PART | The part of the date you want to change. |
| SYSTIMESTAMP | The timestamp precision. |
| TO_CHAR (Dates) | The character string. |
| TO_DATE | The character string. |
| TRUNC (Dates) | The part of the date you want to truncate. |

# TO_CHAR format strings

The TO_CHAR function converts a Date/Time datatype to a string with the format you specify. You can convert the entire date or a part of the date to a string. You might use TO_CHAR to convert dates to string, changing the format for reporting purposes.

TO_CHAR is generally used when the target is a flat file or a database that does not support a Date/Time datatype.

The following table summarizes the format strings for dates in the function TO_CHAR:

| Format string | Description |
|---|---|
| AM, A.M., PM, P.M. | Meridian indicator. Use any of these format strings to specify AM and PM hours. AM and PM return the same values as A.M. and P.M. |
| D | Day of week (1-7), where Sunday equals 1. |
| DD | Day of month (01-31). |
| DDD | Day of year (001-366, including leap years). |
| DAY | Name of day, including up to nine characters (for example, Wednesday). |
| DY | Abbreviated three-character name for a day (for example, Wed). |

| Format string | Description |
|---|---|
| HH, HH12 | Hour of day (01-12). |
| HH24 | Hour of day (00-23), where 00 is 12AM (midnight). |
| J | Modified Julian Day. Converts the calendar date to a string equivalent to its Modified Julian Day value, calculated from Jan 1, 4713 00:00:00 BC. It ignores the time component of the date.<br>For example, the expression TO_CHAR( SHIP_DATE, 'J' ) converts Dec 31 1999 23:59:59 to the string 2451544. |
| MI | Minutes (00-59). |
| MM | Month (01-12). |
| MONTH | Name of month, including up to nine characters (for example, January). |
| MON | Abbreviated three-character name for a month (for example, Jan). |
| Q | Quarter of year (1-4), where January to March equals 1. |
| RR | Last two digits of a year. The function removes the leading digits. For example, if you use 'RR' and pass the year 1997, TO_CHAR returns 97.<br>When used with TO_CHAR, 'RR' produces the same results as, and is interchangeable with, 'YY.' However, when used with TO_DATE, 'RR' calculates the closest appropriate century and supplies the first two digits of the year. |
| SS | Seconds (00-59). |
| SSSSS | Seconds since midnight (00000 - 86399). When you use SSSSS in a TO_CHAR expression, Data Integration only evaluates the time portion of a date.<br>For example, the expression TO_CHAR(SHIP_DATE, 'MM/DD/YYYY SSSSS') converts 12/31/1999 01:02:03 to 12/31/1999 03783. |
| Y | Last digit of a year. The function removes the leading digits.<br>For example, if you use 'Y' and pass the year 1997, TO_CHAR returns 7. |
| YY | Last two digits of a year. The function removes the leading digits.<br>For example, if you use 'YY' and pass the year 1997, TO_CHAR returns 97. |
| YYY | Last three digits of a year. The function removes the leading digits.<br>For example, if you use 'YYY' and pass the year 1997, TO_CHAR returns 997. |
| YYYY | Entire year portion of date. For example, if you use 'YYYY' and pass the year 1997, TO_CHAR returns 1997. |
| W | Week of month (1-5), where week 1 starts on the first day of the month and ends on the seventh, week 2 starts on the eighth day and ends on the fourteenth day. For example, Feb 1 designates the first week of February. |
| WW | Week of year (01-53), where week 01 starts on Jan 1 and ends on Jan 7, week 2 starts on Jan 8 and ends on Jan 14, and so on. |

| Format string | Description |
|---|---|
| - / . ; : | Punctuation that displays in the output. You might use these symbols to separate date parts. For example, you might create the following expression to separate date parts with a period: TO_CHAR( DATES, 'MM.DD.YYYY' ). |
| "text" | Text that displays in the output. For example, if you have the expression: TO_CHAR( DATES, 'MM/DD/YYYY "Sales Were Up"' ) and pass the date Apr 1 1997, the function returns the string '04/01/1997 Sales Were Up'. You can enter multibyte characters that are valid in the repository code page. |
| " " | Use double quotation marks to separate ambiguous format strings, for example D"D""DDD. The empty quotation marks do not appear in the output. |

**Note:** The format string is not case sensitive. It must always be enclosed within single quotation marks.

# Examples

The following examples illustrate the J, SSSSS, RR, and YY format strings. See the individual functions for more examples.

**Note:** The application ignores the time portion of the date in a TO_CHAR expression.

## J format string

Use the J format string in a TO_CHAR expression to convert date values to MJD values expressed as strings. For example:

```
TO_CHAR(SHIP_DATE, 'J')
```

| SHIP_DATE | RETURN_VALUE |
|---|---|
| Dec 31 1999 23:59:59 | 2451544 |
| Jan 1 1900 01:02:03 | 2415021 |

## SSSSS format string

You can also use the format string SSSSS in a TO_CHAR expression. For example, the following expression converts the dates in the SHIP_DATE port to strings representing the total seconds since midnight:

```
TO_CHAR( SHIP_DATE, 'SSSSS')
```

| SHIP_DATE | RETURN_VALUE |
|---|---|
| 12/31/1999 01:02:03 | 3783 |
| 09/15/1996 23:59:59 | 86399 |

### RR format string

The following expression converts dates to strings in the format MM/DD/YY:

```
TO_CHAR( SHIP_DATE, 'MM/DD/RR')
```

| SHIP_DATE | RETURN_VALUE |
|---|---|
| 12/31/1999 01:02:03 | 12/31/99 |
| 09/15/1996 23:59:59 | 09/15/96 |
| 05/17/2003 12:13:14 | 05/17/03 |

### YY format string

In TO_CHAR expressions, the YY format string produces the same results as the RR format string. The following expression converts dates to strings in the format MM/DD/YY:

```
TO_CHAR( SHIP_DATE, 'MM/DD/YY')
```

| SHIP_DATE | RETURN_VALUE |
|---|---|
| 12/31/1999 01:02:03 | 12/31/99 |
| 09/15/1996 23:59:59 | 09/15/96 |
| 05/17/2003 12:13:14 | 05/17/03 |

# TO_DATE and IS_DATE format strings

The TO_DATE function converts a string with the format you specify to a datetime value. TO_DATE is generally used to convert strings from flat files to datetime values. TO_DATE format strings are not internationalized and must be entered in predefined formats as listed in the table below.

**Note:** TO_DATE and IS_DATE use the same set of format strings.

When you create a TO_DATE expression, use a format string for each part of the date in the source string. The source string format and the format string must match, including any date separators. If any parts do not match, Data Integration does not convert the string and skips the row. If you omit the format string, the source string must be in the default date format MM/DD/YYYY HH24:MI:SS.

IS_DATE tells you if a value is a valid date. A valid date is any string representing a valid date in the default date format of MM/DD/YYYY HH24:MI:SS. If the strings you want to test are not in the default date format, use the format strings listed in the following table to specify the date format. If a string does not match the specified format string or is not a valid date, the function returns FALSE (0). If the string matches the format string and is a valid date, the function returns TRUE (1). IS_DATE format strings are not internationalized and must be entered in predefined formats as listed in the following table.

The following table summarizes the format strings for the functions TO_DATE and IS_DATE:

| Format string | Description |
|---|---|
| AM, a.m., PM, p.m. | Meridian indicator. Use any of these format strings to specify AM and PM hours. AM and PM return the same values as do a.m. and p.m. |
| DAY | Name of day, including up to nine characters (for example, Wednesday). The DAY format string is not case sensitive. |
| DD | Day of month (1-31). |
| DDD | Day of year (001-366, including leap years). |
| DY | Abbreviated three-character name for a day (for example, Wed). The DY format string is not case sensitive. |
| HH, HH12 | Hour of day (1-12). |
| HH24 | Hour of day (0-23), where 0 is 12AM (midnight). |
| J | Modified Julian Day. Convert strings in MJD format to date values. It ignores the time component of the source string, assigning all dates the time of 00:00:00.<br>For example, the expression TO_DATE('2451544', 'J') converts 2451544 to Dec 31 1999 00:00:00. |
| MI | Minutes (0-59). |
| MM | Month (1-12). |
| MONTH | Name of month, including up to nine characters (for example, August). Case does not matter. |
| MON | Abbreviated three-character name for a month (for example, Aug). Case does not matter. |
| MS | Milliseconds (0-999). |
| NS | Nanoseconds. TO_DATE and IS_DATE can support sub-seconds by using the format token 'NS.' The unit is nanosecond. If the sub-second portion is in milliseconds, you can still use it by appending three zeroes as shown in the following examples:<br>TO_DATE('2005-05-02 09:23:34 123000', 'YYYY-MM-DD HH24:MI:SS NS')<br>TO_DATE('2005-05-02 09:23:34.123' \|\| '000', 'YYYY-MM-DD HH24:MI:SS.NS')<br>TO_DATE('2005-05-02 09:23:34123000', 'YYYY-MM-DD HH24:MI:SSNS') |
| RR | Four-digit year (for example, 1998, 2034). Use when source strings include two-digit years. Use with TO_DATE to convert two-digit years to four-digit years.<br>- Current Year Between 50 and 99. If the current year is between 50 and 99 (such as 1998) and the year value of the source string is between 0 and 49, Data Integration returns the next century plus the two-digit year from the source string. If the year value of the source string is between 50 and 99, Data Integration returns the current century plus the specified two-digit year.<br>- Current Year Between 0 and 49. If the current year is between 0 and 49 (such as 2003) and the source string year is between 0 and 49, Data Integration returns the current century plus the two-digit year from the source string. If the source string year is between 50 and 99, Data Integration returns the previous century plus the two-digit year from the source string. |
| SS | Seconds (0-59). |

| Format string | Description |
|---|---|
| SSSSS | Seconds since midnight. When you use SSSSS in a TO_DATE expression, Data Integration only evaluates the time portion of a date.<br>For example, the expression TO_DATE( DATE_STR, 'MM/DD/YYYY SSSSS') converts 12/31/1999 3783 to 12/31/1999 01:02:03. |
| US | Microseconds (0-999999). |
| Y | The current year on the machine running the Secure Agent with the last digit of the year replaced with the string value. |
| YY | The current year on the machine running the Secure Agent with the last two digits of the year replaced with the string value. |
| YYY | The current year on the machine running the Secure Agent with the last three digits of the year replaced with the string value. |
| YYYY | Four digits of a year. Do not use this format string if you are passing two-digit years. Use the RR or YY format string instead. |

# Requirements

Data Integration expects the format of the TO_DATE string to meet the following conditions:

- The format of the TO_DATE string must match the format string including any date separators. If it does not, Data Integration might return inaccurate values or skip the row. For example, if you pass the string '20200512', representing May 12, 2020, to TO_DATE, you must include the format string YYYYMMDD. If you do not include a format string, Data Integration expects the string in the default date format MM/DD/YYYY HH24:MI:SS. Likewise, if you pass a string that does not match the format string, Data Integration returns an error and skips the row. For example, if you pass the string 2020120 to TO_DATE and include the format string YYYYMMDD, Data Integration returns an error and skips the row because the string does not match the format string.
- The format string must always be enclosed within single quotation marks.

  **Tip:** By default, Data Integration uses the format string MM/DD/YYYY HH24:MI:SS. The format string is not case sensitive.

# Example

The following examples illustrate the J, RR, and SSSSS format strings. See the individual functions for more examples.

## J format string

The following expression converts strings in the SHIP_DATE_MJD_STRING field to date values in the default date format:

```
TO_DATE (SHIP_DATE_MJD_STR, 'J')
```

| SHIP_DATE_MJD_STR | RETURN_VALUE |
|---|---|
| 2451544 | Dec 31 1999 00:00:00 |

| SHIP_DATE_MJD_STR | RETURN_VALUE |
|---|---|
| 2415021 | Jan 1 1900 00:00:00 |

Because the J format string does not include the time portion of a date, the return values have the time set to 00:00:00.

### RR format string

The following expression converts a string to a four-digit year format. The current year is 1998:

```
TO_DATE( DATE_STR, 'MM/DD/RR')
```

| DATE_STR | RETURN VALUE |
|---|---|
| 04/01/98 | 04/01/1998 00:00:00 |
| 08/17/05 | 08/17/2005 00:00:00 |

### YY format string

The following expression converts a string to a four-digit year format. The current year is 1998:

```
TO_DATE( DATE_STR, 'MM/DD/YY')
```

| DATE_STR | RETURN VALUE |
|---|---|
| 04/01/98 | 04/01/1998 00:00:00 |
| 08/17/05 | 08/17/1905 00:00:00 |

**Note:** For the second row, RR returns the year 2005, but YY returns the year 1905.

### SSSSS format string

The following expression converts strings that include the seconds since midnight to date values:

```
TO_DATE( DATE_STR, 'MM/DD/YYYY SSSSS')
```

| DATE_STR | RETURN_VALUE |
|---|---|
| 12/31/1999 3783 | 12/31/1999 01:02:03 |
| 09/15/1996 86399 | 09/15/1996 23:59:59 |

## Understanding date arithmetic

The transformation language provides built-in date functions so you can perform arithmetic on datetime values as follows:

**ADD_TO_DATE**

Add or subtract a specific portion of a date.

**DATE_DIFF**

Subtract two dates.

**SET_DATE_PART**

Change one part of a date.

You cannot use numeric arithmetic operators (such as + or -) to add or subtract dates.

Data Integration recognizes leap years and accepts dates between Jan. 1, 1753 00:00:00 AD and Dec. 31, 9999 23:59:59 AD.

**Note:** Data Integration uses the Date/Time datatype to specify date values. You can only use the date functions on datetime values.

CHAPTER 5

# Functions

This chapter describes the functions in the transformation language in alphabetical order. Each function description includes:

- Syntax
- Return value
- Example

## Function overview

You can use aggregate functions in mapping tasks. You can use all other functions in mapping tasks and synchronization tasks.

The transformation language provides the following function categories:

- Aggregate
- Conversion
- Data Cleansing
- Date
- Encoding
- Financial
- Horizontal Expansion
- Numerical
- Scientific
- Special
- String
- Test
- Window

### Aggregate functions

Aggregate functions return summary values for non-null values in selected fields.

With aggregate functions you can complete the following tasks:

- Calculate a single value for all rows in a group.

- Return a single value for each group in an Aggregator object.
- Apply filters to calculate values for specific rows in the selected fields.
- Use operators to perform arithmetic within the function.
- Calculate two or more aggregate values derived from the same source columns in a single pass.

Use aggregate functions in mapping tasks only.

The transformation language includes the following aggregate functions:

- AVG
- COUNT
- FIRST
- LAST
- MAX (Date)
- MAX (Number)
- MAX (String)
- MEDIAN
- MIN (Date)
- MIN (Number)
- MIN (String)
- PERCENTILE
- STDDEV
- SUM
- VARIANCE

Use aggregate functions in Aggregator objects only. You can nest only one aggregate function within another aggregate function. Data Integration evaluates the innermost aggregate function expression and uses the result to evaluate the outer aggregate function expression. You cannot nest aggregate functions in advanced mode.

You can set up an Aggregator object that groups by ID and nests two aggregate functions as follows:

```
SUM( AVG( earnings ) )
```

where the dataset contains the following values:

| ID | EARNINGS |
|---|---|
| 1 | 32 |
| 1 | 45 |
| 1 | 100 |
| 2 | 65 |
| 2 | 75 |
| 2 | 76 |
| 3 | 21 |

| ID | EARNINGS |
|---|---|
| 3 | 45 |
| 3 | 99 |

The return value is 186. Data Integration groups by ID, evaluates the AVG expression, and returns three values. Then it adds the values with the SUM function to get the result.

### Filter conditions

Use a filter condition to limit the rows returned in a search.

A filter limits the rows returned in a search. You can apply a filter condition to all aggregate functions and to CUME, MOVINGAVG, and MOVINGSUM. The filter condition must evaluate to TRUE, FALSE, or NULL. If the filter condition evaluates to NULL or FALSE, Data Integration does not select the row.

You can enter any valid transformation expression. For example, the following expression calculates the median salary for all employees who make more than $50,000:

```
MEDIAN( SALARY, SALARY > 50000 )
```

You can also use other numeric values as the filter condition. For example, you can enter the following as the complete syntax for the MEDIAN function, including a numeric field:

```
MEDIAN( PRICE, QUANTITY > 0 )
```

In all cases, Data Integration rounds a decimal value to an integer (for example, 1.5 to 2, 1.2 to 1, 0.35 to 0) for the filter condition. If the value rounds to 0, the filter condition returns FALSE. If you do not want to round up a value, use the TRUNC function to truncate the value to an integer:

```
MEDIAN( PRICE, TRUNC( QUANTITY ) > 0 )
```

If you omit the filter condition, the function selects all rows in the field.

## Conversion functions

The transformation language provides the following conversion functions:

- TO_BIGINT
- TO_CHAR(Date)
- TO_CHAR(Number)
- TO_DATE
- TO_DECIMAL
- TO_FLOAT
- TO_INTEGER

## Data cleansing functions

The transformation language provides a group of functions to eliminate data errors. You can complete the following tasks with data cleansing functions:

- Test source values.
- Convert the datatype of an source value.

- Trim string values.
- Replace characters in a string.
- Encode strings.
- Match patterns in regular expressions.

The transformation language provides the following data cleansing functions:

- BETWEEN
- GREATEST
- IN
- INSTR
- IS_DATE
- IS_NUMBER
- IS_SPACES
- ISNULL
- LEAST
- LTRIM
- METAPHONE
- REG_EXTRACT
- REG_MATCH
- REG_REPLACE
- REPLACECHR
- REPLACESTR
- RTRIM
- SOUNDEX
- SUBSTR
- TO_BIGINT
- TO_CHAR
- TO_DATE
- TO_DECIMAL
- TO_FLOAT
- TO_INTEGER

## Date functions

The transformation language provides a group of date functions to round, truncate, or compare dates, extract one part of a date, or perform arithmetic on a date.

You can pass any value with a date datatype to any of the date functions. However, if you want to pass a string to a date function, you must first use the TO_DATE function to convert it to a transformation Date/Time datatype.

The transformation language provides the following date functions:

- ADD_TO_DATE

- DATE_COMPARE
- DATE_DIFF
- GET_DATE_PART
- LAST_DAY
- MAKE_DATE_TIME
- ROUND
- SET_DATE_PART
- SYSTIMESTAMP
- TRUNC

Several of the date functions include a *format* argument. You must specify one of the transformation language format strings for this argument. Date format strings are not internationalized.

The Date/Time transformation datatype does not support milliseconds. Therefore, if you pass a date with milliseconds, Data Integration truncates the millisecond portion of the date.

## Encoding functions

The transformation language provides the following functions for data encoding, encryption, compression, and checksum:

- AES_DECRYPT
- AES_ENCRYPT
- AES_GCM_DECRYPT
- AES_GCM_ENCRYPT
- COMPRESS
- CRC32
- DEC_BASE64
- DECOMPRESS
- ENC_BASE64
- MD5
- SHA256

## Financial functions

The transformation language provides the following financial functions:

- FV
- NPER
- PMT
- PV
- RATE

## Horizontal expansion functions

Use a horizontal expansion function to create a horizontal macro expression.

Horizontal expansion functions use the following naming convention: %OPR_<function_type>%.

Horizontal expansion functions use square brackets ( [ ] ) instead of parentheses.

The transformation language provides the following horizontal expansion functions:

- %OPR_CONCAT%
- %OPR_CONCATDELIM%
- %OPR_IIF%
- %OPR_SUM%

## Numeric functions

The transformation language provides the following numeric functions:

- ABS
- CEIL
- CONV
- CUME
- EXP
- FLOOR
- LN
- LOG
- MOD
- MOVINGAVG
- MOVINGSUM
- POWER
- RAND
- ROUND
- SIGN
- SQRT
- TRUNC

## Scientific functions

The transformation language provides the following scientific functions:

- COS
- COSH
- SIN
- SINH
- TAN
- TANH

## Special functions

The transformation language provides the following special functions:

- ABORT
- DECODE
- ERROR
- IIF
- SETCOUNTVARIABLE
- SETMAXVARIABLE
- SETMINVARIABLE
- SETVARIABLE

You can nest other functions within special functions.

## String functions

The transformation language provides the following string functions:

- ASCII
- CHOOSE
- CHR
- CHRCODE
- CONCAT
- INDEXOF
- INITCAP
- INSTR
- LENGTH
- LOWER
- LPAD
- LTRIM
- REPLACECHR
- REPLACESTR
- REVERSE
- RPAD
- RTRIM
- SUBSTR
- UPPER

To evaluate character data, the string functions LOWER, UPPER, and INITCAP use the code page of the Secure Agent that runs the task.

## Test functions

The transformation language provides the following test functions:

- ISNULL

- IS_DATE
- IS_NUMBER
- IS_SPACES

# Window functions

In advanced mode, the transformation language includes a group of window functions that perform calculations on a set of rows that are related to the current row. The functions calculate a single return value for every input row.

The transformation language provides the following window functions:

- LAG
- LEAD

You can use window functions in an Expression transformation after you configure the window properties. If you configure window properties, you can also use the aggregate functions as window functions. As window functions, the aggregate functions do not group rows into a single output row but return an output value for each individual row.

## Aggregate functions as window functions

In addition to the LEAD and the LAG functions, you can use aggregate functions as window functions. When you use an aggregate function like SUM or AVG as a window function, you can perform running calculations. Window functions are more flexible than stateful functions because you can set a specific end offset.

To use an aggregate function as a window function, you must define a frame in the window properties to limit the scope of the calculation. The aggregate function performs a calculation across the frame and produces a single value for each row.

### Example

You are a lumber salesperson who sold different quantities of wood over the past two years. You want to calculate a running total of sales quantities.

The following table lists each sale ID, the date, and the quantity sold:

| Sale_ID | Date | Quantity |
|---|---|---|
| 30001 | 2016-08-02 | 10 |
| 10001 | 2016-12-24 | 10 |
| 10005 | 2016-12-24 | 30 |
| 40001 | 2017-01-09 | 40 |
| 10006 | 2017-01-18 | 10 |
| 20001 | 2017-02-12 | 20 |

A SUM function adds all the values and returns one output value. To get a running total for each row, you can define a frame for the function boundaries.

You configure the following properties on the **Window** tab:

- Start offset: All Rows Preceding
- End offset: 0
- Order Key: Date Ascending

You define the following aggregate function:

```
SUM (Quantity)
```

SUM adds the quantity in the current row to the quantities in all the rows preceding the current row. The function returns a running total for each row.

The following table lists a running sum for each date:

| Sale_ID | Date | Quantity | Total |
|---|---|---|---|
| 30001 | 2016-08-02 | 10 | 10 |
| 10001 | 2016-12-24 | 10 | 20 |
| 10005 | 2016-12-24 | 30 | 50 |
| 40001 | 2017-01-09 | 40 | 90 |
| 10006 | 2017-01-18 | 10 | 100 |
| 20001 | 2017-02-12 | 20 | 120 |

## Nested aggregate functions as window functions

A nested aggregate function in a window function performs a separate calculation for each partition.

When you include nested aggregate functions in an Expression transformation and configure the transformation for window functions, the function performs the calculation separately for each partition.

You partition the data by P2 and specify a frame of All Preceding Rows and All Following Rows. The window functions perform the following calculations:

1. COUNT (P1) produces one value for every row. COUNT returns the number of rows in the partition that have non-null values.
2. MEDIAN of that value produces the median of a window of values generated by COUNT.

The window functions produce the following outputs:

| P1 | P2 | Output |
|---|---|---|
| 10 | 1 | 3 |
| 7 | 1 | 3 |
| 12 | 1 | 3 |
| 11 | 2 | 4 |
| 13 | 2 | 4 |
| 8 | 2 | 4 |

| P1 | P2 | Output |
|---|---|---|
| 10 | 2 | 4 |

You can nest aggregate functions with multiple window functions. For example:

```
LAG ( LEAD( MAX( FIRST ( p1 )))
```

**Note:** You can nest multiple window functions LEAD and LAG, but you cannot nest more than one aggregate function within another aggregate function.

# Function quick reference

The following table contains the syntax and a brief description of the functions that can be used in field expressions:

**Note:** The functions that you can use depend on the mapping type.

| Function | Function type | Syntax | Description |
|---|---|---|---|
| %OPR_CONCAT% | Horizontal Expansion | %OPR_CONCAT[ *macro_input_field* ]% | Uses the CONCAT function and expands an expression in an expression macro to concatenate multiple fields.<br>For more information, see "%OPR_CONCAT%" on page 67 |
| %OPR_CONCATDELIM% | Horizontal Expansion | %OPR_CONCATDELIM[ *macro_input_field* ]% | Uses the CONCAT function and expands an expression in an expression macro to concatenate multiple fields, and adds a comma delimiter.<br>For more information, see "%OPR_CONCATDELIM%" on page 68 |
| %OPR_IIF% | Horizontal Expansion | %OPR_IIF[ *condition*, *macro_input_field* [,*value* ] ] % | Uses the IIF function and expands an expression in an expression macro to evaluate a set of IIF statements.<br>For more information, see "%OPR_IIF%" on page 69 |
| %OPR_SUM% | Horizontal Expansion | %OPR_SUM[ *macro_input_field* [, *filter_condition* ] ]% | Uses the SUM function and expands an expression in an expression macro to return the sum of all fields.<br>For more information, see "%OPR_SUM%" on page 70 |
| ABORT | Special | ABORT( *string* ) | Stops the session and issues a specified error message.<br>For more information, see "ABORT" on page 71. |

| Function | Function type | Syntax | Description |
|---|---|---|---|
| ABS | Numeric | ABS( *numeric_value* ) | Returns the absolute value of a numeric value.<br>For more information, see "ABS" on page 71. |
| ADD_TO_DATE | Data Cleansing, Date | ADD_TO_DATE( *date*, *format*, *amount* ) | Adds a specified amount to one part of a date/time value, and returns a date in the same format as the specified date.<br>If you do not specify the year as YYYY, Data Integration assumes the date is in the current century.<br>For more information, see "ADD_TO_DATE" on page 72. |
| AES_DECRYPT | Encoding | AES_DECRYPT ( *value, key* ) | Returns the decrypted value in string format, after performing AES-ECB decryption on the input value.<br>For more information, see "AES_DECRYPT" on page 75. |
| AES_ENCRYPT | Encoding | AES_ENCRYPT ( *value, key* ) | Returns binary data in encrypted format, after performing AES-ECB encryption on the input value.<br>For more information, see "AES_ENCRYPT" on page 76. |
| AES_GCM_DECRYPT | Encoding | AES_GCM_DECRYPT ( *value, init_vector, key* [, *keysize* ] ) | Returns plaintext, a decrypted value as a string, after performing AES-GCM decryption on an input value with the given initialization vector and key.<br>For more information, see "AES_GCM_DECRYPT" on page 77. |
| AES_GCM_ENCRYPT | Encoding | AES_GCM_ENCRYPT ( *value, init_vector, key* [, *keysize* ] ) | Returns ciphertext as a binary value after performing AES-GCM encryption on an input value with the given initialization vector and key. The ciphertext is encrypted plaintext.<br>For more information, see "AES_GCM_ENCRYPT" on page 78. |
| ASCII | String | ASCII ( *string* ) | Returns the numeric ASCII value of the first character of the string passed to the function.<br>This function is identical in behavior to the CHRCODE function. If you use the ASCII function in existing expressions, it will still work correctly. However, when you create new expressions, use the CHRCODE function instead of the ASCII function.<br>For more information, see "ASCII" on page 79. |
| AVG | Aggregate | AVG ( *numeric_value* [, *filter_condition* ] ) | Returns the average of all values in a group of rows.<br>For more information, see "AVG" on page 80. |

| Function | Function type | Syntax | Description |
|---|---|---|---|
| CEIL | Numeric | CEIL ( *numeric_value* ) | Returns the smallest integer greater than or equal to the specified numeric value.<br>For more information, see "CEIL" on page 81. |
| CHOOSE | String | CHOOSE( *index, string1*, [*string2*, ..., *stringN*] ) | Chooses a string from a list of strings based on a given position.<br>For more information, see "CHOOSE" on page 82. |
| CHR | String | CHR( *numeric_value* ) | Returns the ASCII character corresponding to the specified numeric value.<br>For more information, see "CHR" on page 83. |
| CHRCODE | String | CHRCODE( *string* ) | Returns the numeric ASCII value of the first character of the string passed to the function.<br>This function is identical in behavior to the ASCII function.<br>For more information, see "CHRCODE" on page 84. |
| COMPRESS | Encoding | COMPRESS( *value* ) | Compresses data using the zlib compression algorithm.<br>For more information, see "COMPRESS" on page 85. |
| CONCAT | String | CONCAT( *first_string*, *second_string* ) | Concatenates two strings.<br>For more information, see "CONCAT" on page 85. |
| CONVERT_BASE | Numeric | CONVERT_BASE( *value, source_base, dest_base )* | Converts a number from one base value to another base value.<br>For more information, see "CONVERT_BASE" on page 87. |
| COS | Scientific | COS( *numeric_value* ) | Returns the cosine of a numeric value (expressed in radians).<br>For more information, see "COS" on page 87. |
| COSH | Scientific | COSH( *numeric_value* ) | Returns the hyperbolic cosine of a numeric value (expressed in radians).<br>For more information, see "COSH" on page 88. |
| COUNT | Aggregate | COUNT( *value* [, *filter_condition*] )<br>or<br>COUNT( * [, *filter_condition*] ) | Returns the number of rows that have non-null values in a group.<br>For more information, see "COUNT" on page 89. |

| Function | Function type | Syntax | Description |
|---|---|---|---|
| CRC32 | Encoding | CRC32( *value* ) | Returns a 32-bit Cyclic Redundancy Check (CRC32) value.<br>For more information, see "CRC32" on page 91. |
| CUME | Numeric | CUME( *numeric_value* [, *filter_condition*] ) | Returns a running total.<br>For more information, see "CUME" on page 92. |
| DATE_COMPARE | Data Cleansing, Date | DATE_COMPARE( *date1*, *date2* ) | Returns a value indicating the earlier of two dates.<br>For more information, see "DATE_COMPARE" on page 93. |
| DATE_DIFF | Data Cleansing, Date | DATE_DIFF( *date1*, *date2*, *format* ) | Returns the length of time between two dates, measured in the specified increment (years, months, days, hours, minutes, or seconds).<br>For more information, see "DATE_DIFF" on page 94. |
| DEC_BASE64 | Encoding | DEC_BASE64( *value* ) | Decodes the value and returns a string with the binary data representation of the data.<br>For more information, see "DEC_BASE64" on page 96. |
| DECODE | Special | DECODE( *value*, *first_search*, *first_result* [, *second_search*, *second_result* ]...[, *default* ] ) | Searches a column for the specified value.<br>For more information, see "DECODE" on page 97. |
| DECOMPRESS | Encoding | DECOMPRESS( *value*, *precision* ) | Decompresses data using the zlib compression algorithm.<br>For more information, see "DECOMPRESS" on page 99. |
| ENC_BASE64 | Encoding | ENC_BASE64( *value* ) | Encodes data by converting binary data to string data using Multipurpose Internet Mail Extensions (MIME) encoding.<br>For more information, see "ENC_BASE64" on page 100. |
| ERROR | Special | ERROR( *string* ) | Causes the Data Integration to skip a row. It writes the row into the error rows file with the specified error message.<br>For more information, see "ERROR" on page 100. |
| EXP | Numeric | EXP( *exponent* ) | Returns e raised to the specified power (exponent), where e=2.71828183.<br>For more information, see "EXP" on page 101. |

| Function | Function type | Syntax | Description |
|---|---|---|---|
| FIRST | Aggregate | FIRST( *value* [, *filter_condition* ] ) | Returns the first value found within a field or group.<br>For more information, see "FIRST" on page 102. |
| FLOOR | Numeric | FLOOR( *numeric_value* ) | Returns the largest integer less than or equal to the specified numeric value.<br>For more information, see "FLOOR" on page 103. |
| FV | Financial | FV( *rate, terms, payment* [, *present value, type*] ) | Returns the future value of an investment, where you make periodic, constant payments and the investment earns a constant interest rate.<br>For more information, see "FV" on page 104. |
| GET_DATE_PART | Date, Data Cleansing | GET_DATE_PART( *date, format* ) | Returns the specified part of a date as an integer value, based on the default date format of MM/DD/YYYY HH24:MI:SS.<br>For more information, see "GET_DATE_PART" on page 105. |
| GREATEST | Data Cleansing | GREATEST( *value1*, [*value2*, ..., *valueN*,] *CaseFlag* ) | Returns the greatest value from a list of input values.<br>For more information, see "GREATEST" on page 107. |
| IIF | Special | IIF( *condition, value2* [, *value2* ] ) | Returns one of two values you specify, based on the results of a condition.<br>For more information, see "IIF" on page 108. |
| IN | Data Cleansing | IN( *valueToSearch*, *value1*, [*value2*, ..., *valueN*,] *CaseFlag* ) | Matches input data to a list of values.<br>For more information, see "IN " on page 110. |
| INDEXOF | String | INDEXOF( *valueToSearch, string1*, [*string2*, ..., *stringN*,] *CaseFlag* ) | Finds the index of a string among a list of strings.<br>For more information, see "INDEXOF" on page 110. |
| INITCAP | String | INITCAP( *string* ) | Capitalizes the first letter in each word of a string and converts all other letters to lowercase.<br>For more information, see "INITCAP" on page 111. |
| INSTR | String, Data Cleansing | INSTR( *string*, *search_value* [, *start* [, *occurrence* ] ] ) | Returns the position of a character set in a string, counting from left to right.<br>For more information, see "INSTR" on page 113. |

| Function | Function type | Syntax | Description |
|---|---|---|---|
| IS_DATE | Data Cleansing, Test | IS_DATE( *value* ) | Returns whether a value is a valid date.<br>For more information, see "IS_DATE" on page 115. |
| IS_NUMBER | Data Cleansing, Test | IS_NUMBER( *value* ) | Returns whether a string is a valid number.<br>For more information, see "IS_NUMBER" on page 117. |
| IS_SPACES | Data Cleansing, Test | IS_SPACES( *value* ) | Returns whether a value consists entirely of spaces.<br>For more information, see "IS_SPACES" on page 119. |
| ISNULL | Data Cleansing, Test | ISNULL( *value* ) | Returns whether a value is NULL.<br>For more information, see "ISNULL" on page 120. |
| LAG | Window | LAG( field_name, offset, default_value ) | Returns the value from a preceding row.<br>For more information, see "LAG" on page 121. |
| LAST | Aggregate | LAST( *value* [, *filter_condition* ] ) | Returns the last row in the selected field.<br>For more information, see "LAST" on page 123. |
| LAST_DAY | Data Cleansing, Date | LAST_DAY( *date* ) | Returns the date of the last day of the month for each date in a column.<br>For more information, see "LAST_DAY" on page 123. |
| LEAD | Window | LEAD( field_name, offset, default_value ) | Returns the value from a following row.<br>For more information, see "LEAD" on page 125. |
| LEAST | Data Cleansing | LEAST( *value1*, [*value2*, ..., *valueN*,] *CaseFlag* ) | Returns the smallest value from a list of input values.<br>For more information, see "LEAST" on page 127. |
| LENGTH | String | LENGTH( *string* ) | Returns the number of characters in a string, including trailing blanks.<br>For more information, see "LENGTH" on page 128. |
| LN | Numeric | LN( *numeric_value* ) | Returns the natural logarithm of a numeric value.<br>For more information, see "LN" on page 128. |
| LOG | Numeric | LOG( *base, exponent* ) | Returns the logarithm of a numeric value.<br>For more information, see "LOG" on page 129. |

| Function | Function type | Syntax | Description |
|---|---|---|---|
| LOWER | String | LOWER( *string* ) | Converts uppercase string characters to lowercase.<br>For more information, see "LOWER" on page 131. |
| LPAD | String | LPAD( *first_string*, *length* [, *second_string* ] ) | Adds a set of blanks or characters to the beginning of a string to set a string to a specified length.<br>For more information, see "LPAD" on page 132. |
| LTRIM | String, Data Cleansing | LTRIM( *string* [, *trim_set* ] ) | Removes blanks or characters from the beginning of a string.<br>For more information, see "LTRIM" on page 133. |
| MAKE_DATE_TIME | Data Cleansing, Date | MAKE_DATE_TIME( *year, month, day, hour, minute, second* ) | Returns the date and time based on the input values.<br>For more information, see "MAKE_DATE_TIME" on page 134. |
| MAX (Dates) | Aggregate | MAX( *date* [, *filter_condition*] ) | Returns the latest date found within a field or group.<br>For more information, see "MAX (Dates)" on page 135. |
| MAX (Numbers) | Aggregate | MAX( *numeric_value* [, *filter_condition*] ) | Returns the maximum numeric value found within a field or group.<br>For more information, see "MAX (Numbers)" on page 136. |
| MAX (String) | Aggregate | MAX( *string* [, *filter_condition*] ) | Returns the highest string value found within a field or group.<br>For more information, see "MAX (String)" on page 138. |
| MD5 | Encoding | MD5( *value* ) | Calculates the checksum of the input value. The function uses Message-Digest algorithm 5 (MD5).<br>For more information, see "MD5" on page 139. |
| MEDIAN | Aggregate | MEDIAN( *numeric_value* [, *filter_condition* ] ) | Returns the median of all values in a selected field.<br>For more information, see "MEDIAN" on page 140. |
| METAPHONE | Data Cleansing | METAPHONE( *string* [, *length*] ) | Encodes characters of the English language alphabet (A-Z).<br>For more information, see "METAPHONE" on page 141. |

| Function | Function type | Syntax | Description |
|---|---|---|---|
| MIN (Dates) | Aggregate | MIN( *date* [, *filter_condition*] ) | Returns the earliest date found in a field or group.<br>For more information, see "MIN (Dates)" on page 145. |
| MIN (Numbers) | Aggregate | MIN( *date* [, *filter_condition*] ) | Returns the smallest numeric value found in a field or group.<br>For more information, see "MIN (Numbers)" on page 146. |
| MIN (String) | Aggregate | MIN( *string* [, *filter_condition*] ) | Returns the lowest string value found in a field or group.<br>For more information, see "MIN (String)" on page 147. |
| MOD | Numeric | MOD( *numeric_value, divisor* ) | Returns the remainder of a division calculation.<br>For more information, see "MOD" on page 149. |
| MOVINGAVG | Numeric | MOVINGAVG( *numeric_value, rowset* [, *filter_condition*] ) | Returns the average (row-by-row) of a specified set of rows.<br>For more information, see "MOVINGAVG" on page 150. |
| MOVINGSUM | Numeric | MOVINGSUM( *numeric_value, rowset* [, *filter_condition*] ) | Returns the sum (row-by-row) of a specified set of rows.<br>For more information, see "MOVINGSUM" on page 151. |
| NPER | Financial | NPER( *rate, present value, payment* [, *future value, type*]) | Returns the number of periods for an investment based on a constant interest rate and periodic, constant payments.<br>For more information, see "NPER" on page 152. |
| PERCENTILE | Aggregate | PERCENTILE( *numeric_value, percentile* [, *filter_condition* ] ) | Calculates the value that falls at a given percentile in a group of numbers.<br>For more information, see "PERCENTILE" on page 153. |
| PMT | Financial | PMT( *Rate, terms, present value* [, *future value, type*] ) | Returns the payment for a loan based on constant payments and a constant interest rate.<br>For more information, see "PMT" on page 155. |
| POWER | Numeric | POWER( *base, exponent* ) | Returns a value raised to the specified exponent.<br>For more information, see "POWER" on page 156. |
| PV | Financial | PV( *Rate, terms, payment* [, *future value, type*] ) | Returns the present value of an investment.<br>For more information, see "PV" on page 157. |

| Function | Function type | Syntax | Description |
|---|---|---|---|
| RAND | Numeric | RAND( *seed* ) | Returns a random number between 0 and 1.<br>For more information, see "RAND" on page 158. |
| RATE | Financial | RATE( *terms, payment, present value* [, *future value, type*] ) | Returns the interest rate earned per period by a security.<br>For more information, see "RATE" on page 158. |
| REG_EXTRACT | Data Cleansing | REG_EXTRACT( *subject, pattern, subPatternNum* ) | Extracts subpatterns of a regular expression within an input value.<br>For more information, see "REG_EXTRACT" on page 159. |
| REG_MATCH | Data Cleansing | REG_MATCH( *subject, pattern* ) | Returns whether a value matches a regular expression pattern.<br>For more information, see "REG_MATCH" on page 161. |
| REG_REPLACE | Data Cleansing | REG_REPLACE( subject, pattern, replace, numReplacements ) | Replaces characters in a string with a another character pattern.<br>For more information, see "REG_REPLACE" on page 163. |
| REPLACECHR | String, Data Cleansing | REPLACECHR( *CaseFlag, InputString, OldCharSet, NewChar* ) | Replaces characters in a string with a single character or no character.<br>For more information, see "REPLACECHR" on page 164. |
| REPLACESTR | String, Data Cleansing | REPLACESTR ( *InputString, OldString1*, [OldString2, ... OldStringN,] *NewString* ) | Replaces characters in a string with a single character, multiple characters, or no character.<br>For more information, see "REPLACESTR" on page 167. |
| REVERSE | String | REVERSE( *string* ) | Reverses the input string.<br>For more information, see "REVERSE" on page 169. |
| ROUND | Data Cleansing, Date, Numeric | ROUND( *date* [, *format* ] )<br>or<br>ROUND( *numeric_value* [, *precision* ] ) | For data cleansing, rounds one part of a date. For numeric values, rounds numbers to a specified digit.<br>For more information, see "ROUND (Dates)" on page 170 or "ROUND (Numbers)" on page 173. |
| RPAD | String | RPAD( *first_string*, *length* [, *second_string* ] ) | Converts a string to a specified length by adding blanks or characters to the end of the string.<br>For more information, see "RPAD" on page 175. |

| Function | Function type | Syntax | Description |
|---|---|---|---|
| RTRIM | String, Data Cleansing | RTRIM( *string* [, *trim_set* ] ) | Removes blanks or characters from the end of a string.<br>For more information, see "RTRIM" on page 176. |
| SET_DATE_PART | Data Cleansing, Date | SET_DATE_PART( *date, format, value* ) | Sets one part of a date/time value to a specified value.<br>For more information, see "SET_DATE_PART" on page 177. |
| SETCOUNTVARIABLE | Special | SETCOUNTVARIABLE( *$$Variable* ) | Counts the rows evaluated by the function and increments the current value of an in-out parameter based on the count.<br>For more information, see "SETCOUNTVARIABLE" on page 180. |
| SETMAXVARIABLE | Special | SETMAXVARIABLE( $$*Variable*, *value* ) | Sets the current value of an in-out parameter to the higher of two values: the current value of the parameter or the value you specify.<br>For more information, see "SETMAXVARIABLE" on page 180. |
| SETMINVARIABLE | Special | SETMINVARIABLE( $$*Variable*, *value* ) | Sets the current value of an in-out parameter to the lower of two values: the current value of the parameter or the value you specify.<br>For more information, see "SETMINVARIABLE" on page 181. |
| SETVARIABLE | Special | SETVARIABLE( $$*Variable*, *value* ) | Sets the current value of an in-out parameter to a value you specify.<br>For more information, see "SETVARIABLE" on page 182. |
| SHA256 | Encoding | SHA256( *value* ) | Returns the SHA-256 digest of the input value.<br>For more information, see "SHA256" on page 183. |
| SIGN | Numeric | SIGN( *numeric_value* ) | Indicates whether a numeric value is positive, negative, or 0.<br>For more information, see "SIGN" on page 184. |
| SIN | Scientific | SIN( *numeric_value* ) | Returns the sin of a numeric value expressed in radians.<br>For more information, see "SIN" on page 185. |
| SINH | Scientific | SINH( *numeric_value* ) | Returns the hyperbolic sin of a numeric value expressed in radians.<br>For more information, see "SINH" on page 186. |

| Function | Function type | Syntax | Description |
|---|---|---|---|
| SOUNDEX | Data Cleansing | SOUNDEX( *string*) | Encodes a string value into a four-character string.<br>For more information, see "SOUNDEX" on page 187. |
| SQRT | Numeric | SQRT( *numeric_value* ) | Returns the square root of a positive numeric value.<br>For more information, see "SQRT" on page 188. |
| STDDEV | Aggregate | STDDEV( *numeric_value* [,*filter_condition*] ) | Returns the standard deviation of the numeric values you pass to this function.<br>For more information, see "STDDEV" on page 189. |
| SUBSTR | String, Data Cleansing | SUBSTR( *string*, *start* [, *length* ] ) | Returns a portion of a string.<br>For more information, see "SUBSTR" on page 191. |
| SUM | Aggregate | SUM( *numeric_value* [, *filter_condition* ] ) | Returns the sum of all values in the selected field.<br>For more information, see "SUM" on page 193. |
| SYSTIMESTAMP | Date | SYSTIMESTAMP( [*format*] ) | Returns the current date and time with precision to the nanosecond of the system that hosts the Secure Agent that starts the task.<br>For more information, see "SYSTIMESTAMP" on page 194. |
| TAN | Scientific | TAN( *numeric_value* ) | Returns the tangent of a numeric value expressed in radians.<br>For more information, see "TAN" on page 195. |
| TANH | Scientific | TANH( *numeric_value* ) | Returns the hyperbolic tangent of a numeric value expressed in radians.<br>For more information, see "TANH" on page 196. |
| TO_BIGINT | Conversion, Data Cleansing | TO_BIGINT( *value* [, *flag*] ) | Converts a string or numeric value to a bigint value.<br>For more information, see "TO_BIGINT" on page 197. |
| TO_CHAR | Conversion, Data Cleansing | TO_CHAR( *date* [, *format* ] )<br>or<br>TO_CHAR( *numeric_value* ) | Converts dates or numeric values to text strings.<br>For more information, see "TO_CHAR (Dates)" on page 198 or "TO_CHAR (Numbers)" on page 202. |

| Function | Function type | Syntax | Description |
|---|---|---|---|
| TO_DATE | Conversion, Data Cleansing | TO_DATE( *string* [, *format* ] ) | Converts a character string to a date datatype in the same format as the character string. For conversion, you must specify the date format if the string is not in the mm/dd/yyyy hh:mi:ss format.<br>For more information, see "TO_DATE" on page 203. For more information about date formats, see Chapter 4, "Dates" on page 29. |
| TO_DECIMAL | Conversion, Data Cleansing | TO_DECIMAL( *value* [, *scale* ] ) | Converts any value (except binary) to a decimal.<br>For more information, see "TO_DECIMAL" on page 206. |
| TO_FLOAT | Conversion, Data Cleansing | TO_FLOAT( *value* ) | Converts any value (except binary) to a double-precision floating point number (the Double datatype).<br>For more information, see "TO_FLOAT" on page 207. |
| TO_INTEGER | Conversion, Data Cleansing | TO_INTEGER( *value* ) | Converts any value (except binary) to an integer by rounding the decimal portion of a value.<br>For more information, see "TO_INTEGER" on page 207. |
| TRUNC | Data Cleansing, Date, Numeric | TRUNC( *date* [, *format* ] )<br>or<br>TRUNC( *numeric_value* [, *precision*]) | Truncates dates to a specific year, month, day, hour, or minute.<br>Truncates numeric values to a specific digit.<br>For more information, see "TRUNC (Dates)" on page 209 or "TRUNC (Numbers)" on page 211. |
| UPPER | String | UPPER( *string* ) | Converts lowercase string characters to uppercase.<br>For more information, see "UPPER" on page 213. |
| VARIANCE | Aggregate | VARIANCE( *numeric_value* [, *filter_condition* ] ) | Returns the variance of a value you pass to it.<br>For more information, see "VARIANCE" on page 213. |

# Functions in advanced mode

In advanced mode, the expression editor includes the functions that enable advanced functionality. When you copy a mapping to advanced mode, you might need to update expressions to use the available functions.

When you copy a mapping to advanced mode, the **Validation** panel shows validation errors if a function is not available in advanced mode. To resolve the error, update the data logic in the mapping to use only the functions that are available in advanced mode. Functions in advanced mode might behave differently.

The following table lists the functions and the mappings where they are available:

| Function | Availability |
|---|---|
| %OPR_CONCAT% | Available in all mappings |
| %OPR_CONCATDELIM% | Available in all mappings |
| %OPR_IIF% | Available in all mappings |
| %OPR_SUM% | Available in all mappings |
| ABORT | Available outside of advanced mode |
| ABS | Available in all mappings |
| ADD_TO_DATE | Available in all mappings |
| AES_DECRYPT | Available in all mappings |
| AES_ENCRYPT | Available in all mappings |
| AES_GCM_DECRYPT | Available in all mappings |
| AES_GCM_ENCRYPT | Available in all mappings |
| ASCII | Available in all mappings |
| AVG | Available in all mappings |
| CEIL | Available in all mappings |
| CHOOSE | Available in all mappings |
| CHR | Available in all mappings |
| CHRCODE | Available in all mappings |
| COMPRESS | Available in all mappings |
| CONCAT | Available in all mappings |
| CONVERT_BASE | Available in all mappings |
| COS | Available in all mappings |
| COSH | Available in all mappings |
| COUNT | Available in all mappings |
| CRC32 | Available in all mappings |
| CUME | Available outside of advanced mode |
| DATE_COMPARE | Available in all mappings |
| DATE_DIFF | Available in all mappings |

| Function | Availability |
|---|---|
| DEC_BASE64 | Available in all mappings |
| DECODE | Available in all mappings |
| DECOMPRESS | Available in all mappings |
| ENC_BASE64 | Available in all mappings |
| ERROR | Available outside of advanced mode |
| EXP | Available in all mappings |
| FIRST | Available outside of advanced mode<br>Available in advanced mode only as a window function |
| FLOOR | Available in all mappings |
| FV | Available in all mappings |
| GET_DATE_PART | Available in all mappings |
| GREATEST | Available in all mappings |
| IIF | Available in all mappings |
| IN | Available in all mappings |
| INDEXOF | Available in all mappings |
| INITCAP | Available in all mappings |
| INSTR | Available in all mappings |
| IS_DATE | Available in all mappings |
| IS_NUMBER | Available in all mappings |
| IS_SPACES | Available in all mappings |
| ISNULL | Available in all mappings |
| LAG | Available in advanced mode |
| LAST | Available outside of advanced mode<br>Available in advanced mode only as a window function |
| LAST_DAY | Available in all mappings |
| LEAD | Available in advanced mode |
| LEAST | Available in all mappings |
| LENGTH | Available in all mappings |

| Function | Availability |
|---|---|
| LN | Available in all mappings |
| LOG | Available in all mappings |
| LOWER | Available in all mappings |
| LPAD | Available in all mappings |
| LTRIM | Available in all mappings |
| MAKE_DATE_TIME | Available in all mappings |
| MAX (Dates) | Available in all mappings |
| MAX (Numbers) | Available in all mappings |
| MAX (String) | Available in all mappings |
| MD5 | Available in all mappings |
| MEDIAN | Available in all mappings |
| METAPHONE | Available in all mappings |
| MIN (Dates) | Available in all mappings |
| MIN (Numbers) | Available in all mappings |
| MIN (String) | Available in all mappings |
| MOD | Available in all mappings |
| MOVINGAVG | Available outside of advanced mode |
| MOVINGSUM | Available outside of advanced mode |
| NPER | Available in all mappings |
| PERCENTILE | Available in all mappings |
| PMT | Available in all mappings |
| POWER | Available in all mappings |
| PV | Available in all mappings |
| RAND | Available in all mappings |
| RATE | Available in all mappings |
| REG_EXTRACT | Available in all mappings |
| REG_MATCH | Available in all mappings |

| Function | Availability |
|---|---|
| REG_REPLACE | Available in all mappings |
| REPLACECHR | Available in all mappings |
| REPLACESTR | Available in all mappings |
| REVERSE | Available in all mappings |
| ROUND | Available in all mappings |
| RPAD | Available in all mappings |
| RTRIM | Available in all mappings |
| SET_DATE_PART | Available in all mappings |
| SETCOUNTVARIABLE | Available in all mappings |
| SETMAXVARIABLE | Available in all mappings |
| SETMINVARIABLE | Available in all mappings |
| SETVARIABLE | Available outside of advanced mode |
| SHA256 | Available outside of advanced mode |
| SIGN | Available in all mappings |
| SIN | Available in all mappings |
| SINH | Available in all mappings |
| SOUNDEX | Available in all mappings |
| SQRT | Available in all mappings |
| STDDEV | Available in all mappings |
| SUBSTR | Available in all mappings |
| SUM | Available in all mappings |
| SYSTIMESTAMP | Available in all mappings |
| TAN | Available in all mappings |
| TANH | Available in all mappings |
| TO_BIGINT | Available in all mappings |
| TO_CHAR | Available in all mappings |
| TO_DATE | Available in all mappings |

| Function | Availability |
|---|---|
| TO_DECIMAL | Available in all mappings |
| TO_FLOAT | Available in all mappings |
| TO_INTEGER | Available in all mappings |
| TRUNC | Available in all mappings |
| UPPER | Available in all mappings |
| VARIANCE | Available in all mappings |

# %OPR_CONCAT%

Uses the CONCAT function and expands an expression in an expression macro to concatenate multiple fields. %OPR_CONCAT% converts all data to text, and then concatenates the fields.

## Syntax

```
%OPR_CONCAT[ macro_input_field ]%
```

The following table describes the argument for this function:

| Argument | Required/ Optional | Description |
|---|---|---|
| *macro_input_field* | Required | Any datatype except Binary. You can use a set of incoming fields or a set of constants. You can enter a macro input field or an expression that includes at least one macro input field. |

## Return Value

String.

NULL if all strings in the set of fields are NULL.

Returns NULL if all strings are null. Otherwise, ignores the null values and concatenates the strings that are not null.

## Example

You have two legacy systems that have merged into one system. You need to concatenate the identification numbers from both systems into another system. The macro input field %ClientID% contains the following fields:

```
ID1, ID2
```

The following expression concatenates the fields in %ClientID%:

```
 %OPR_CONCAT[ %ClientID% ]%
```

%OPR_CONCAT% concatenates the following IDs into one return value:

| ID1 | ID2 | RETURN VALUE |
|---|---|---|
| A6JU4199 | 7021 | A6JU41997021 |

| ID1 | ID2 | RETURN VALUE |
|---|---|---|
| NULL | 7022 | 7022 |
| T7QX9018 | 7023 | T7QX90187023 |
| C2CL5421 | 7024 | C2CL54217024 |
| NULL | NULL | NULL |

# %OPR_CONCATDELIM%

Uses the CONCAT function and expands an expression in an expression macro to concatenate multiple fields, and adds a comma delimiter. %OPR_CONCATDELIM% converts all data to text, and then concatenates the fields.

## Syntax

```
%OPR_CONCATDELIM[ macro_input_field ]%
```

The following table describes the argument for this function:

| Argument | Required/ Optional | Description |
|---|---|---|
| *macro_input_field* | Required | Any datatype except Binary. You can use a set of incoming fields or a set of constants. You can enter a macro input field or an expression that includes at least one macro input field. |

## Return Value

String.

Returns NULL if all strings are null. Otherwise, ignores the null values and concatenates the strings that are not null.

If all strings in the set are NULL, %OPR_CONCATDELIM% returns NULL.

## Example

The macro input field %Address% contains the following fields:

```
City, State, Zip
```

The following expression concatenates the fields in %Address% and adds a comma between fields:

```
%OPR_CONCATDELIM[ %Address% ]%
```

%OPR_CONCAT_DELIM% concatenates the following address fields into one return value:

| CITY | STATE | ZIP | RETURN VALUE |
|---|---|---|---|
| Reston | VA | 20190 | Reston,VA,20190 |
| Maywood | IL | NULL | Maywood,IL |
| Eagle River | AK | 99577 | Eagle River,AK,99577 |
| St. Louis | MO | 63110 | St. Louis,MO,63110 |
| NULL | NULL | NULL | NULL |

# %OPR_IIF%

Uses the IIF function and expands an expression in an expression macro to evaluate a set of IIF statements.

## Syntax

```
%OPR_IIF[ condition, macro_input_field [,value ]]%
```

The following table describes the arguments for this function:

| Argument | Required/ Optional | Description |
|---|---|---|
| *condition* | Required | The condition you want to evaluate. You can enter any valid expression that evaluates to TRUE or FALSE. You can enter a macro input field or an expression that includes at least one macro input field. |
| *macro_input_field* | Required | The values you want to return if the condition is TRUE. Any data type except Binary. The return value is the data type specified by this argument. You can enter a macro input field or an expression that includes at least one macro input field. |
| *value* | Optional | The value you want to return if the condition is FALSE. Any data type except Binary. In a horizontal or vertical macro, you can enter any valid expression. In a hybrid macro, enter the input macro field that is used to define the output macro field. |

Unlike conditional functions in some systems, the FALSE (*value*) condition in the %OPR_IIF% function is not required. If you omit *value*, the function returns the following values when the condition is FALSE:

- Zero if *macro_input_field* is a Numeric data type.
- Empty string if *macro_input_field* is a String data type.
- NULL if *macro_input_field* is a Date/Time data type.

## Return Value

*macro_input_field* if the condition is TRUE.

*value* if the condition is FALSE.

If the data contains multibyte characters and the condition argument compares string data, the return value depends on the code page of the Secure Agent that runs the task.

## %OPR_IIF% and Data types

When you use %OPR_IIF%, the data type of the return value is the same as the data type of the result with the greatest precision.

When at least one result is Double, the data type of the return value is Double.

### Example

The following macro input fields %tmin% and %tmax% create ranges that correspond to the following %type% macro input field:

| %tmin% | %tmax% | %type% |
|---|---|---|
| 0 | 30 | very cold |
| 31 | 60 | cold |
| 61 | 90 | warm |
| 91 | 100 | hot |
| 101 | 150 | very hot |

The following expression assigns a type based on the temperature data:

```
%OPR_IIF [ (TEMP >= %tmin%) AND (TEMP <= %tmax%), '%type%', 'out of range' ]%
```

%OPR_IIF% evaluates the following temperatures and returns a type:

| TEMP | RETURN VALUE |
|---|---|
| 79 | warm |
| 21 | very cold |
| 170 | out of range |
| 99 | hot |
| 90 | warm |

# %OPR_SUM%

Uses the SUM function and expands an expression in an expression macro to return the sum of all fields. Optionally, you can apply a filter to limit the rows you read to calculate the total. You can use only one other aggregate function within SUM, and the nested function must return a numeric datatype.

### Syntax

```
%OPR_SUM[ macro_input_field [, filter_condition ] ]%
```

The following table describes the arguments for this function:

| Argument | Required/ Optional | Description |
|---|---|---|
| *macro _input _field* | Required | Numeric datatype. Passes the values you want to add. You can enter a macro input field or an expression that includes at least one macro input field. |
| *filter_condition* | Optional | Limits the rows in the search. The filter condition must be a numeric value or evaluate to TRUE, FALSE, or NULL. You can enter any valid transformation expression. |

### Return Value

Numeric value.

NULL if all fields passed to the function are NULL or if no rows are selected. For example, if a filter condition evaluates to FALSE or NULL for all rows, %OPR_SUM% returns NULL .

If a single value is NULL, %OPR_SUM% ignores it. If all values are NULL, %OPR_SUM% returns NULL.

### Group By

%OPR_SUM% groups values based on group by fields that you define in the transformation, and returns one result for each group.

If there is no group by field, %OPR_SUM% treats all rows as one group, returning one value.

### Example

The %Sales% macro input field represents the following fields:

```
Q1, Q2, Q3, Q4
```

The following expression returns the sum of the fields that %Sales% represents:

```
%OPR_SUM[ %Sales% ]%
```

%OPR_SUM% adds the following sales figures from each quarter to return one value:

| Q1 | Q2 | Q3 | Q4 | RETURN VALUE |
|---|---|---|---|---|
| 50 | 200 | 50 | 100 | 400 |
| 100 | NULL | 100 | 250 | 450 |
| 500 | 200 | 200 | 200 | 1100 |
| NULL | NULL | NULL | NULL | NULL |
| 400 | NULL | NULL | 400 | 800 |

# ABORT

Stops the task and issues an error message in the job details. When Data Integration encounters an ABORT function, it stops transforming data at that row.

Use ABORT to validate data. Generally, you use ABORT within an IIF or DECODE function to set rules for aborting a task. You might use ABORT to avoid writing null values to the target.

### Syntax

```
ABORT( string )
```

| Argument | Required/ Optional | Description |
|---|---|---|
| string | Required | String. The message you want to display in the job details when the session stops. The string can be any length. You can enter any valid expression. |

### Return Value

NULL.

# ABS

Returns the absolute value of a numeric value.

### Syntax

```
ABS( numeric_value )
```

| Argument | Required/ Optional | Description |
|---|---|---|
| *numeric_value* | Required | Numeric datatype. Returns the absolute value of a number. You can enter any valid expression. |

### Return Value

Positive numeric value. ABS returns the same datatype as the numeric value passed as an argument. If you pass a Double, it returns a Double. Likewise, if you pass an Integer, it returns an Integer.

NULL if you pass a null value to the function.

### Example

The following expression returns the difference between two numbers as a positive value, regardless of which number is larger:

```
ABS( PRICE - COST )
```

| PRICE | COST | RETURN VALUE |
|---|---|---|
| 250 | 150 | 100 |
| 52 | 48 | 4 |
| 169.95 | 69.95 | 100 |
| 59.95 | NULL | NULL |
| 70 | 30 | 40 |
| 430 | 330 | 100 |
| 100 | 200 | 100 |

# ADD_TO_DATE

Adds a specified amount to one part of a datetime value, and returns a date in the same format as the date you pass to the function.

ADD_TO_DATE accepts positive and negative integer values. Use ADD_TO_DATE to change the following parts of a date:

**Year**

Enter a positive or negative integer in the *amount* argument. Use any of the year format strings: Y, YY, YYY, or YYYY. For example, the expression `ADD_TO_DATE ( SHIP_DATE, 'YY', 10 )` adds 10 years to all dates in the SHIP_DATE column.

**Month**

Enter a positive or negative integer in the *amount* argument. Use any of the month format strings: MM, MON, MONTH. For example, the expression `ADD_TO_DATE( SHIP_DATE, 'MONTH', -10 )` subtracts 10 months from each date in the SHIP_DATE column.

**Day**

Enter a positive or negative integer in the *amount* argument. Use any of the day format strings: D, DD, DDD, DY, and DAY. For example, the expression `ADD_TO_DATE( SHIP_DATE, 'DD', 10 )` adds 10 days to each date in the SHIP_DATE column.

**Hour**

Enter a positive or negative integer in the *amount* argument. Use any of the hour format strings: HH, HH12, HH24. For example, the expression `ADD_TO_DATE( SHIP_DATE, 'HH', 14 )` adds 14 hours to each date in the SHIP_DATE column.

**Minute**

Enter a positive or negative integer in the *amount* argument. Use the MI format string to set the minute. For example, the expression `ADD_TO_DATE( SHIP_DATE, 'MI', 25 )` adds 25 minutes to each date in the SHIP_DATE column.

**Seconds**

Enter a positive or negative integer in the *amount* argument. Use the SS format string to set the second. For example, the following expression adds 59 seconds to each date in the SHIP_DATE column:

```
ADD_TO_DATE( SHIP_DATE, 'SS', 59 )
```

## Syntax

```
ADD_TO_DATE( date, format, amount )
```

| Argument | Required/ Optional | Description |
|---|---|---|
| *date* | Required | Date/Time datatype. Passes the values you want to change. You can enter any valid expression. |
| *format* | Required | A format string specifying the portion of the date value you want to change. Enclose the format string within single quotation marks, for example, 'mm'. The format string is not case sensitive. |
| *amount* | Required | An integer value specifying the amount of years, months, days, hours, and so on by which you want to change the date value. You can enter any valid expression that evaluates to an integer. |

## Return Value

Date in the same format as the date you pass to this function.

NULL if a null value is passed as an argument to the function.

## Example

The following expressions all add one month to each date in the DATE_SHIPPED column. If you pass a value that creates a day that does not exist in a particular month, Data Integration returns the last day of the month. For example, if you add one month to Jan 31 1998, Data Integration returns Feb 28 1998.

ADD_TO_DATE recognizes leap years and adds one month to Jan 29 2000:

```
ADD_TO_DATE( DATE_SHIPPED, 'MM', 1 )
ADD_TO_DATE( DATE_SHIPPED, 'MON', 1 )
ADD_TO_DATE( DATE_SHIPPED, 'MONTH', 1 )
```

| DATE_SHIPPED | RETURN VALUE | |
|---|---|---|
| Jan 12 1998 12:00:30AM | Feb 12 1998 12:00:30AM | |
| Jan 31 1998 6:24:45PM | Feb 28 1998 6:24:45PM | |
| Jan 29 2000 5:32:12AM | Feb 29 2000 5:32:12AM | *(Leap Year)* |
| Oct 9 1998 2:30:12PM | Nov 9 1998 2:30:12PM | |
| NULL | NULL | |

The following expressions subtract 10 days from each date in the DATE_SHIPPED column:

```
ADD_TO_DATE( DATE_SHIPPED, 'D', -10 )
ADD_TO_DATE( DATE_SHIPPED, 'DD', -10 )
ADD_TO_DATE( DATE_SHIPPED, 'DDD', -10 )
ADD_TO_DATE( DATE_SHIPPED, 'DY', -10 )
ADD_TO_DATE( DATE_SHIPPED, 'DAY', -10 )
```

| DATE_SHIPPED | RETURN VALUE | |
|---|---|---|
| Jan 1 1997 12:00:30AM | Dec 22 1996 12:00AM | |
| Jan 31 1997 6:24:45PM | Jan 21 1997 6:24:45PM | |
| Mar 9 1996 5:32:12AM | Feb 29 1996 5:32:12AM | *(Leap Year)* |
| Oct 9 1997 2:30:12PM | Sep 30 1997 2:30:12PM | |
| Mar 3 1996 5:12:20AM | Feb 22 1996 5:12:20AM | |
| NULL | NULL | |

The following expressions subtract 15 hours from each date in the DATE_SHIPPED column:

```
ADD_TO_DATE( DATE_SHIPPED, 'HH', -15 )
ADD_TO_DATE( DATE_SHIPPED, 'HH12', -15 )
ADD_TO_DATE( DATE_SHIPPED, 'HH24', -15 )
```

| DATE_SHIPPED | RETURN VALUE | |
|---|---|---|
| Jan 1 1997 12:00:30AM | Dec 31 1996 9:00:30AM | |
| Jan 31 1997 6:24:45PM | Jan 31 1997 3:24:45AM | |
| Oct 9 1997 2:30:12PM | Oct 8 1997 11:30:12PM | |
| Mar 3 1996 5:12:20AM | Mar 2 1996 2:12:20PM | |
| Mar 1 1996 5:32:12AM | Feb 29 1996 2:32:12PM | *(Leap Year)* |
| NULL | NULL | |

### Working with Dates

Use the following tips when working with ADD_TO_DATE:

- You can add or subtract any part of the date by specifying a format string and making the *amount* argument a positive or negative integer.
- If you pass a value that creates a day that does not exist in a particular month, Data Integration returns the last day of the month. For example, if you add one month to Jan 31 1998, Data Integration returns Feb 28 1998.
- You can nest TRUNC and ROUND to manipulate dates.
- You can nest TO_DATE to convert strings to dates.
- ADD_TO_DATE changes only one portion of the date, which you specify. If you modify a date so that it changes from standard to daylight savings time, you need to change the hour portion of the date.

# AES_DECRYPT

Returns the decrypted value in string format. Data Integration uses the Advanced Encryption Standard (AES) algorithm with the Electronic Codebook (ECB) mode of operation. The AES algorithm is a FIPS-approved cryptographic algorithm that uses 128-bit encryption.

## Syntax

```
AES_DECRYPT ( value, key )
```

| Argument | Required/ Optional | Description |
|---|---|---|
| value | Required | Binary datatype. Value you want to decrypt. |
| key | Required | String datatype. Precision of 16 characters or fewer. Use the same key to decrypt a value that you used to encrypt it. |

## Return Value

Decrypted string value.

NULL if the input value is a null value.

## Example

The following example returns decrypted Social Security numbers. In this example, Data Integration derives the key from the first three numbers of the Social Security number using the SUBSRT function:

```
AES_DECRYPT( SSN_ENCRYPT, SUBSTR( SSN,1,3 ))
```

| SSN_ENCRYPT | DECRYPTED VALUE |
|---|---|
| 07FB945926849D2B1641E708C85E4390 | 832-17-1672 |
| 9153ACAB89D65A4B81AD2ABF151B099D | 832-92-4731 |
| AF6B5E4E39F974B3F3FB0F22320CC60B | 832-46-7552 |
| 992D6A5D91E7F59D03B940A4B1CBBCBE | 832-53-6194 |

```
SSN_ENCRYPT                                    DECRYPTED VALUE

992D6A5D91E7F59D03B940A4B1CBBCBE               832-81-9528
```

# AES_ENCRYPT

Returns binary data in encrypted format. Data Integration uses the Advanced Encryption Standard (AES) algorithm with the Electronic Codebook (ECB) mode of operation. The AES algorithm is a FIPS-approved cryptographic algorithm that uses 128-bit encryption.

Use this function to prevent sensitive data from being visible to everyone. For example, to store Social Security numbers in a database, use the AES_ENCRYPT function to encrypt the Social Security numbers to maintain confidentiality.

## Syntax

```
AES_ENCRYPT ( value, key )
```

| Argument | Required/ Optional | Description |
|---|---|---|
| value | Required | String datatype. Value you want to encrypt. |
| key | Required | String datatype. Precision of 16 characters or fewer. |

## Return Value

Encrypted binary value.

NULL if the input is a null value.

## Example

The following example returns encrypted values for Social Security numbers. In this example, Data Integration derives the key from the first three numbers of the Social Security number using the SUBSTR function:

```
AES_ENCRYPT( SSN, SUBSTR( SSN,1,3 ))
```

```
SSN                  ENCRYPTED VALUE

832-17-1672          07FB945926849D2B1641E708C85E4390

832-92-4731          9153ACAB89D65A4B81AD2ABF151B099D

832-46-7552          AF6B5E4E39F974B3F3FB0F22320CC60B

832-53-6194          992D6A5D91E7F59D03B940A4B1CBBCBE

832-81-9528          992D6A5D91E7F59D03B940A4B1CBBCBE
```

## Tip

If the target does not support binary data, use AES_ENCRYPT with the ENC_BASE64 function to store the data in a format that is compatible with the database.

# AES_GCM_DECRYPT

Returns plaintext as a string. Data Integration uses the Advanced Encryption Standard (AES) algorithm with the Galois/Counter Mode (GCM) of operation. The AES algorithm is a FIPS-approved cryptographic algorithm that uses 128, 192, or 256-bit keys.

**Note:** The **Validate** button doesn't validate this function. Review the syntax and argument rules to ensure that the arguments are valid.

## Syntax

```
AES_GCM_DECRYPT ( value, init_vector, key [, keysize] )
```

| Argument | Required/ Optional | Description |
|---|---|---|
| value | Required | Binary data type. The ciphertext value to be decrypted into plaintext. |
| init_vector | Required | String data type. Use the same initialization vector (IV) to decrypt a value that you used to encrypt the ciphertext. The IV must be 96-bit and randomly generated. |
| key | Required | String data type of size 128, 192, or 256 bits. Use the same key to decrypt a value that you used to encrypt it. |
| keysize | The size of the key argument, in bits, determines whether the keysize argument is optional or required.<br>- If the key argument size is <= 128 bits, then the keysize is optional.<br>- If the key argument size is > 128 bits and <= 192 bits, then the keysize is 192 bits, and required.<br>- If the key argument size is >192 and <=256 bits, then the keysize is 256 bits, and required. | Integer data type. Size of the key provided.<br>Possible values:128, 192, 256 bits. Default value is 128 bits. |

## Return Value

Decrypted string plaintext.

NULL if the input value is a null value.

## Example

The following examples return decrypted Social Security numbers. The *init_vector* and *key* are the same values that were used to encrypt the Social Security number.

In this example, the *init_vector* is 12 characters, or 96 bits; the *key* is 16 characters, or 128 bits; the *keysize* is optional because the key is the default value of 128 bits:

```
AES_GCM_DECRYPT(SSN_ENCRYPT, '012345678901', '1234567890123456', 128)
```

In this example, the *init_vector* is 12 characters, or 96 bits; the *key* is 17 characters, or 136 bits; the *keysize* is required because the key is > 128 bits. Because the size of the key is less than 192 bits, the key is padded with null characters:

```
AES_GCM_DECRYPT(SSN_ENCRYPT, '123456789012', '12345678901234567', 192)
```

# AES_GCM_ENCRYPT

Returns binary ciphertext. Data Integration uses the Advanced Encryption Standard (AES) algorithm with the Galois/Counter Mode (GCM) of operation. The AES algorithm is a FIPS-approved cryptographic algorithm that uses 128, 192, or 256-bit keys.

Use this function to prevent sensitive data from being visible to everyone. For example, to store Social Security numbers in a database, use the AES_GCM_ENCRYPT function to encrypt the Social Security numbers to maintain confidentiality.

AES-GCM creates an authentication tag of 128 bits and appends it to encrypted ciphertext. Decryption verifies and removes the authentication tag. An authentication tag is a cryptographic checksum on data that reveals both accidental errors and the intentional modification of the data.

**Note:** The **Validate** button doesn't validate this function. Review the syntax and argument rules to ensure that the arguments are valid.

## Syntax

```
AES_GCM_ENCRYPT ( value, init_vector, key [, keysize] )
```

| Argument | Required/ Optional | Description |
|---|---|---|
| value | Required | String data type. The plaintext value to be encrypted into ciphertext. |
| init_vector | Required | String data type. Use the initialization vector (IV) to encrypt the plaintext. The IV must be 96-bit and randomly generated. The IV is a block of bits that is used along with the key during encryption to add randomness to the start of the encryption process.<br>**Note:** Do not re-use the IV with the same key and a different encryption string. |
| key | Required | String data type of size 128, 192, or 256 bits. If the size of the key is less than the keysize, the remainder is padded as null. |
| keysize | The size of the key argument, in bits, determines whether the keysize argument is optional or required.<br>- If the key argument size is <= 128 bits, then the keysize is optional.<br>- If the key argument size is > 128 bits and <= 192 bits, then the keysize is 192 bits, and required.<br>- If the key argument size is >192 and <=256 bits, then the keysize is 256 bits, and required. | Integer data type. Size of the key provided.<br>Possible values:128, 192, or 256 bits. Default value is 128 bits. |

## Return Value

Encrypted binary ciphertext.

NULL if the input is a null value.

## Example

The following examples return encrypted values for a Social Security number. The *init_vector* and *key* values will be used later to decrypt the Social Security number.

In this example, the *init_vector* is 12 characters, or 96 bits; the *key* is 16 characters, or 128 bits; the *keysize* is optional because the key is the default value of 128 bits:

```
AES_GCM_ENCRYPT('832-17-1672', '012345678901', '1234567890123456', 128)
```

In this example, the *init_vector* is 12 characters, or 96 bits; the *key* is 17 characters, or 136 bits; the *keysize* is required. Because the size of the key is less than 192 bits, the key is padded with null characters:

```
AES_GCM_ENCRYPT('832-17-1672', '123456789012', '12345678901234567', 192)
```

### Tip

If the target does not support binary data, use AES_GCM_ENCRYPT with the ENC_BASE64 function to store the data in a format that is compatible with the database.

# ASCII

The ASCII function returns the numeric UNICODE value of the first character of the string passed to the function.

You can pass a string of any size to ASCII, but it evaluates only the first character in the string. Before you pass any string value to ASCII, you can parse out the specific character you want to convert to a UNICODE value. For example, you might use RTRIM or another string-manipulation function. If you pass a numeric value, ASCII converts it to a character string and returns the UNICODE value of the first character in the string.

**Note:** This function is identical in behavior to the CHRCODE function. If you use ASCII in existing expressions, they will still work correctly. However, when you create new expressions, use the CHRCODE function instead of the ASCII function.

### Syntax

```
ASCII ( string )
```

| Argument | Required/ Optional | Description |
|---|---|---|
| *string* | Required | Character string. Passes the value you want to return as an UNICODE value. You can enter any valid expression. |

### Return Value

Integer. The UNICODE value of the first character in the string.

NULL if a value passed to the function is NULL.

### Example

The following expression returns the UNICODE value for the first character of each value in the ITEMS column:

```
ASCII( ITEMS )
```

| ITEMS | RETURN VALUE |
|---|---|
| Flashlight | 70 |
| Compass | 67 |

| ITEMS | RETURN VALUE |
|---|---|
| Safety Knife | 83 |
| Depth/Pressure Gauge | 68 |
| Regulator System | 82 |

# AVG

Returns the average of all values in a group of rows. Optionally, you can apply a filter to limit the rows you read to calculate the average.

You can nest only one other aggregate function within AVG, and the nested function must return a numeric data type. You cannot nest aggregate functions in advanced mode.

Use only in mapping tasks.

## Syntax

```
AVG( numeric_value [, filter_condition ] )
```

| Argument | Required/ Optional | Description |
|---|---|---|
| *numeric_value* | Required | Numeric data type. Passes the values for which you want to calculate an average. You can enter any valid transformation expression. |
| *filter_condition* | Optional | Limits the rows in the search. The filter condition must be a numeric value or evaluate to TRUE, FALSE, or NULL. You can enter any valid transformation expression. |

## Return Value

Numeric value.

NULL if all values passed to the function are NULL or if no rows are selected. For example, the filter condition evaluates to FALSE or NULL for all rows.

## Nulls

If a value is NULL, AVG ignores the row. However, if all values passed are NULL, AVG returns NULL.

## Group By

AVG groups values based on group by fields you define in the transformation, returning one result for each group.

If there is not a group by field, AVG treats all rows as one group, returning one value.

### Example

The following expression returns the average wholesale cost of flashlights:

```
AVG( WHOLESALE_COST, ITEM_NAME='Flashlight' )
```

| ITEM_NAME | WHOLESALE_COST |
|---|---|
| Flashlight | 35.00 |
| Navigation Compass | 8.05 |
| Regulator System | 150.00 |
| Flashlight | 29.00 |
| Depth/Pressure Gauge | 88.00 |
| Flashlight | 31.00 |

**RETURN VALUE:** 31.66

### Tip

You can perform arithmetic on the values passed to AVG before the function calculates the average. For example:

```
AVG( QTY * PRICE - DISCOUNT )
```

# CEIL

Returns the smallest integer greater than or equal to the numeric value passed to this function. For example, if you pass 3.14 to CEIL, the function returns 4. If you pass 3.98 to CEIL, the function returns 4. Likewise, if you pass -3.17 to CEIL, the function returns -3.

### Syntax

```
CEIL( numeric_value )
```

| Argument | Required/ Optional | Description |
|---|---|---|
| *numeric_value* | Required | Numeric datatype. You can enter any valid expression. |

### Return Value

Integer if you pass a numeric value with declared precision between 0 and 28.

Double value if you pass a numeric value with declared precision greater than 28.

NULL if a value passed to the function is NULL.

### Example

The following expression returns the price rounded to the next integer:

```
CEIL( PRICE )
```

| PRICE | RETURN VALUE |
|---|---|
| 39.79 | 40 |
| 125.12 | 126 |
| 74.24 | 75 |
| NULL | NULL |
| -100.99 | -100 |

### Tip

You can perform arithmetic on the values passed to CEIL before CEIL returns the next integer value. For example, if you wanted to multiply a numeric value by 10 before you calculated the smallest integer less than the modified value, you might write the function as follows:

```
CEIL( PRICE * 10 )
```

# CHOOSE

Chooses a string from a list of strings based on a given position. You specify the position and the value. If the value matches the position, Data Integration returns the value.

### Syntax

```
CHOOSE( index, string1, [string2, ..., stringN] )
```

| Argument | Required/ Optional | Description |
|---|---|---|
| index | Required | Numeric datatype. Enter a number based on the position of the value you want to match. |
| string | Required | Any character value. |

### Return Value

The string that matches the position of the index value.

NULL if no string matches the position of the index value.

### Example

The following expression returns the string ‘flashlight’ based on an index value of 2:

```
CHOOSE( 2, 'knife', 'flashlight', 'diving hood' )
```

The following expression returns NULL based on an index value of 4:

```
CHOOSE( 4, 'knife', 'flashlight', 'diving hood' )
```

CHOOSE returns NULL because the expression does not contain a fourth argument.

# CHR

CHR returns the ASCII character corresponding to the numeric value you pass to this function. ASCII values fall in the range 0 to 255. You can pass any integer to CHR, but only ASCII codes 32 to 126 are printable characters.

## Syntax

```
CHR( numeric_value )
```

| Argument | Required/ Optional | Description |
|---|---|---|
| numeric_value | Required | Numeric datatype. The value you want to return as an ASCII character. You can enter any valid expression. |

## Return Value

ASCII character. A string containing one character.

NULL if a value passed to the function is NULL.

## Example

The following expression returns the ASCII character for each numeric value in the ITEM_ID column:

```
CHR( ITEM_ID )
```

| ITEM_ID | RETURN VALUE |
|---|---|
| 65 | A |
| 122 | z |
| NULL | NULL |
| 88 | X |
| 100 | d |
| 71 | G |

Use the CHR function to concatenate a single quotation mark onto a string. The single quotation mark is the only character that you cannot use inside a string literal. Consider the following example:

```
'Joan' || CHR(39) || 's car'
```

The return value is:

```
Joan's car
```

# CHRCODE

CHRCODE returns the numeric UNICODE value of the first character of the string passed to the function.

Normally, before you pass any string value to CHRCODE, you parse out the specific character you want to convert to a UNICODE value. For example, you might use RTRIM or another string-manipulation function. If you pass a numeric value, CHRCODE converts it to a character string and returns the UNICODE value of the first character in the string.

**Note:** This function is identical in behavior to the ASCII function. If you currently use ASCII in expressions, it will still work correctly. However, when you create new expressions, use the CHRCODE function instead of the ASCII function.

## Syntax

```
CHRCODE ( string )
```

| Argument | Required/ Optional | Description |
|---|---|---|
| *string* | Required | Character string. Passes the values you want to return as UNICODE values. You can enter any valid expression. |

## Return Value

Integer. The UNICODE value of the first character in the string.

NULL if a value passed to the function is NULL.

## Example

The following expression returns the UNICODE value for the first character of each value in the ITEMS column:

```
CHRCODE( ITEMS )
```

| ITEMS | RETURN VALUE |
|---|---|
| Flashlight | 70 |
| Compass | 67 |
| Safety Knife | 83 |
| Depth/Pressure Gauge | 68 |
| Regulator System | 82 |

# COMPRESS

Compresses data using the zlib compression algorithm. The zlib compression algorithm is compatible with WinZip. Use the COMPRESS function before you send large amounts of data over a wide area network.

## Syntax

```
COMPRESS( value )
```

| Argument | Required/ Optional | Description |
|---|---|---|
| *value* | Required | String datatype. Data that you want to compress. |

## Return Value

Compressed binary value of the input value.

NULL if the input is a null value.

## Example

Your organization has an online order service. You want to send customer order data over a wide area network. The source contains a row that is 10 MB. You can compress the data in this row using COMPRESS. When you compress the data, you decrease the amount of data Data Integration writes over the network. As a result, you may improve task performance.

# CONCAT

Concatenates two strings. CONCAT converts all data to text before concatenating the strings. Alternatively, use the || string operator to concatenate strings. Using the || string operator instead of CONCAT improves performance when you run tasks.

## Syntax

```
CONCAT( first_string, second_string )
```

| Argument | Required/ Optional | Description |
|---|---|---|
| *first_string* | Required | Any datatype except Binary. The first part of the string you want to concatenate. You can enter any valid expression. |
| *second_string* | Required | Any datatype except Binary. The second part of the string you want to concatenate. You can enter any valid expression. |

## Return Value

String.

NULL if both string values are NULL.

## Nulls

If one of the strings is NULL, CONCAT ignores it and returns the other string.

If both strings are NULL, CONCAT returns NULL.

## Example

The following expression concatenates the names in the FIRST_NAME and LAST_NAME columns:

```
CONCAT( FIRST_NAME, LAST_NAME )
```

| FIRST_NAME | LAST_NAME | RETURN VALUE |
|---|---|---|
| John | Baer | JohnBaer |
| NULL | Campbell | Campbell |
| Bobbi | Apperley | BobbiApperley |
| Jason | Wood | JasonWood |
| Dan | Covington | DanCovington |
| Greg | NULL | Greg |
| NULL | NULL | NULL |
| 100 | 200 | 100200 |

CONCAT does not add spaces to separate strings. If you want to add a space between two strings, you can write an expression with two nested CONCAT functions. For example, the following expression first concatenates a space on the end of the first name and then concatenates the last name:

```
CONCAT( CONCAT( FIRST_NAME, ' ' ), LAST_NAME )
```

| FIRST_NAME | LAST_NAME | RETURN VALUE |
|---|---|---|
| John | Baer | John Baer |
| NULL | Campbell | Campbell *(includes leading space)* |
| Bobbi | Apperley | Bobbi Apperley |
| Jason | Wood | Jason Wood |
| Dan | Covington | Dan Covington |
| Greg | NULL | Greg |
| NULL | NULL | NULL |

Use the CHR and CONCAT functions to concatenate a single quotation mark onto a string. The single quotation mark is the only character you cannot use inside a string literal. Consider the following example:

```
CONCAT( 'Joan', CONCAT( CHR(39), 's car' ))
```

The return value is:

```
Joan's car
```

# CONVERT_BASE

Converts a number from one base value to another base value.

## Syntax

```
CONVERT_BASE( value, source_base, dest_base )
```

| Argument | Required/ Optional | Description |
|---|---|---|
| value | Required | String datatype. Value you want to convert from one base to another base. |
| source_base | Required | Numeric datatype. Current base value of the data you want to convert. Minimum base is 2. Maximum base is 36. |
| *dest_base* | Required | Numeric datatype. Base value you want to convert the data to. Minimum base is 2. Maximum base is 36. |

## Return Value

Numeric value.

## Example

The following example converts 2222 from the decimal base value 10 to the binary base value 2:

```
CONVERT_BASE( 2222, 10, 2 )
```

Data Integration returns 100010101110.

# COS

Returns the cosine of a numeric value (expressed in radians).

## Syntax

```
COS( numeric_value )
```

| Argument | Required/ Optional | Description |
|---|---|---|
| numeric_value | Required | Numeric datatype. Numeric data expressed in radians (degrees multiplied by pi divided by 180). Passes the values for which you want to calculate a cosine. You can enter any valid expression. |

## Return Value

Double value.

NULL if a value passed to the function is NULL.

### Example

The following expression returns the cosine for all values in the DEGREES column:

```
COS( DEGREES * 3.14159265359 / 180 )
```

| DEGREES | RETURN VALUE |
|---|---|
| 0 | 1.0 |
| 90 | 0.0 |
| 70 | 0.342020143325593 |
| 30 | 0.866025403784421 |
| 5 | 0.996194698091745 |
| 18 | 0.951056516295147 |
| 89 | 0.0174524064371813 |
| NULL | NULL |

### Tip

You can perform arithmetic on the values passed to COS before the function calculates the cosine. For example, you can convert the values in the column to radians before calculating the cosine, as follows:

```
COS( ARCS * 3.14159265359 / 180 )
```

# COSH

Returns the hyperbolic cosine of a numeric value (expressed in radians).

### Syntax

```
COSH( numeric_value )
```

| Argument | Required/ Optional | Description |
|---|---|---|
| numeric_value | Required | Numeric datatype. Numeric data expressed in radians (degrees multiplied by pi divided by 180). Passes the values for which you want to calculate the hyperbolic cosine. You can enter any valid expression. |

### Return Value

Double value.

NULL if a value passed to the function is NULL.

### Example

The following expression returns the hyperbolic cosine for the values in the ANGLES column:

```
COSH( ANGLES )
```

| ANGLES | RETURN VALUE |
|---|---|
| 1.0 | 1.54308063481524 |
| 2.897 | 9.0874465864177 |
| 3.66 | 19.4435376920294 |
| 5.45 | 116.381231106176 |
| 0 | 1.0 |
| 0.345 | 1.06010513656773 |
| NULL | NULL |

### Tip

You can perform arithmetic on the values passed to COSH before the function calculates the hyperbolic cosine. For example:

```
COSH( MEASURES.ARCS / 360 )
```

# COUNT

Returns the number of rows that have non-null values in a group. Optionally, you can include the asterisk (*) argument to count all input values in a transformation. You can apply a condition to filter rows before counting them.

You can nest only one other aggregate function within COUNT. You cannot nest aggregate functions in advanced mode.

Use only in mapping tasks.

### Syntax

```
COUNT( value [, filter_condition] )
```

or

```
COUNT( * [, filter_condition] )
```

| Argument | Required/ Optional | Description |
|---|---|---|
| *value* | Required | Any data type except binary. Passes the values you want to count. You can enter any valid transformation expression. |
| * | Optional | Use to count *all rows* in a transformation. |
| *filter_condition* | Optional | Limits the rows in the search. The filter condition must be a numeric value or evaluate to TRUE, FALSE, or NULL. You can enter any valid transformation expression. |

## Return Value

Integer.

0 if all values passed to this function are NULL (unless you include the asterisk argument).

## Nulls

If all values are NULL, the function returns 0.

If you apply the asterisk argument, this function counts all rows, regardless if a column in a row contains a null value.

If you apply the *value* argument, this function ignores columns with null values.

## Group By

COUNT groups values based on group by fields you define in the transformation, returning one result for each group. If there is no group by field COUNT treats all rows as one group, returning one value.

## Examples

The following expression counts the items with less than 5 quantity in stock, excluding null values:

```
COUNT( ITEM_NAME, IN_STOCK < 5 )
```

| ITEM_NAME | IN_STOCK |
|---|---|
| Flashlight | 10 |
| NULL | 2 |
| Compass | NULL |
| Regulator System | 5 |
| Safety Knife | 8 |
| Halogen Flashlight | 1 |

**RETURN VALUE:** 1

In this example, the function counted the Halogen flashlight but not the NULL item. The function counts all rows in a transformation, including null values, as illustrated in the following example:

```
COUNT( *, QTY < 5 )
```

| ITEM_NAME | QTY |
|---|---|
| Flashlight | 10 |
| NULL | 2 |
| Compass | NULL |
| Regulator System | 5 |
| Safety Knife | 8 |
| Halogen Flashlight | 1 |

**RETURN VALUE:** 2

In this example, the function counts the NULL item and the Halogen Flashlight. If you include the asterisk argument, but do not use a filter, the function counts all rows that pass into the transformation. For example:

```
COUNT( * )
```

| ITEM_NAME | QTY |
|---|---|
| Flashlight | 10 |
| NULL | 2 |
| Compass | NULL |
| Regulator System | 5 |
| Safety Knife | 8 |
| Halogen Flashlight | 1 |

**RETURN VALUE:** 6

# CRC32

Returns a 32-bit Cyclic Redundancy Check (CRC32) value. Use CRC32 to find data transmission errors. You can also use CRC32 if you want to verify that data stored in a file has not been modified.

**Note:** CRC32 can return the same output for different input strings. If you use CRC32 to generate keys, you may receive unexpected results.

### Syntax

```
CRC32( value )
```

| Argument | Required/ Optional | Description |
|---|---|---|
| value | Required | String or Binary datatype. Passes the values you want to perform a redundancy check on. Input value is case sensitive. The case of the input value affects the return value. For example, CRC32(informatica) and CRC32 (Informatica) return different values. |

### Return Value

32-bit integer value.

### Example

You want to read data from a source across a wide area network. You want to make sure the data has been modified during transmission. You can compute the checksum for the data in the file and store it along with the file. When Data Integration reads the source data, Data Integration can use CRC32 to compute the checksum and compare it to the stored value. If the two values are the same, the data has not been modified.

## CUME

Returns a running total. A running total means CUME returns a total each time it adds a value. You can add a condition to filter rows out of the row set before calculating the running total.

Use CUME and similar functions, such as MOVINGAVG and MOVINGSUM, to simplify reporting by calculating running values.

### Syntax

```
CUME( numeric_value [, filter_condition] )
```

| Argument | Required/ Optional | Description |
|---|---|---|
| *numeric_value* | Required | Numeric datatype. Passes the values for which you want to calculate a running total. You can enter any valid expression. You can create a nested expression to calculate a running total based on the results of the function as long as the result is a numeric value. |
| *filter_condition* | Optional | Limits the rows in the search. The filter condition must be a numeric value or evaluate to TRUE, FALSE, or NULL. You can enter any valid expression. |

### Return Value

Numeric value.

NULL if all values passed to the function are NULL, or if no rows are selected (for example, the filter condition evaluates to FALSE or NULL for all rows).

### Nulls

If a value is NULL, CUME returns the running total for the previous row. However, if all values in the selected column are NULL, CUME returns NULL.

## Examples

The following example returns the running total of the PERSONAL_SALES column:

```
CUME( PERSONAL_SALES )
```

| PERSONAL_SALES | RETURN VALUE |
|---|---|
| 40000 | 40000 |
| 80000 | 120000 |
| 40000 | 160000 |
| 60000 | 220000 |
| NULL | 220000 |
| 50000 | 270000 |

Likewise, you can add values before calculating a running total:

```
CUME( CA_SALES + OR_SALES )
```

| CA_SALES | OR_SALES | RETURN VALUE |
|---|---|---|
| 40000 | 10000 | 50000 |
| 80000 | 50000 | 180000 |
| 40000 | 2000 | 222000 |
| 60000 | NULL | 222000 |
| NULL | NULL | 222000 |
| 50000 | 3000 | 275000 |

# DATE_COMPARE

Returns an integer indicating which of two dates is earlier. DATE_COMPARE returns an integer value rather than a date value.

## Syntax

```
DATE_COMPARE( date1, date2 )
```

| Argument | Required/ Optional | Description |
|---|---|---|
| *date1* | Required | Date/Time datatype. The first date you want to compare. You can enter any valid expression as long as it evaluates to a date. |
| *date2* | Required | Date/Time datatype. The second date you want to compare. You can enter any valid expression as long as it evaluates to a date. |

### Return Value

-1 if the first date is earlier.

0 if the two dates are equal.

1 if the second date is earlier.

NULL if one of the date values is NULL.

### Example

The following expression compares each date in the DATE_PROMISED and DATE_SHIPPED columns, and returns an integer indicating which date is earlier:

```
DATE_COMPARE( DATE_PROMISED, DATE_SHIPPED )
```

| DATE_PROMISED | DATE_SHIPPED | RETURN VALUE |
|---|---|---|
| Jan 1 1997 | Jan 13 1997 | -1 |
| Feb 1 1997 | Feb 1 1997 | 0 |
| Dec 22 1997 | Dec 15 1997 | 1 |
| Feb 29 1996 | Apr 12 1996 | -1 *(Leap year)* |
| NULL | Jan 6 1997 | NULL |
| Jan 13 1997 | NULL | NULL |

# DATE_DIFF

Returns the length of time between two dates. You can request the format to be years, months, days, hours, minutes, or seconds. Data Integration subtracts the second date from the first date and returns the difference.

### Syntax

```
DATE_DIFF( date1, date2, format )
```

| Argument | Required/ Optional | Description |
|---|---|---|
| date1 | Required | Date/Time datatype. Passes the values for the first date you want to compare. You can enter any valid expression. |
| date2 | Required | Date/Time datatype. Passes the values for the second date you want to compare. You can enter any valid expression. |
| *format* | Required | Format string specifying the date or time measurement. You can specify years, months, days, hours, minutes, or seconds. You can specify only one part of the date, such as 'mm'. Enclose the format strings within single quotation marks. The format string is not case sensitive. For example, the format string 'mm' is the same as 'MM', 'Mm' or 'mM'. |

## Return Value

Double value. If *date1* is later than *date2*, the return value is a positive number. If *date1* is earlier than *date2*, the return value is a negative number.

0 if the dates are the same.

NULL if one (or both) of the date values is NULL.

## Example

The following expressions return the number of hours between the DATE_PROMISED and DATE_SHIPPED columns:

```
DATE_DIFF( DATE_PROMISED, DATE_SHIPPED, 'HH' )
DATE_DIFF( DATE_PROMISED, DATE_SHIPPED, 'HH12' )
DATE_DIFF( DATE_PROMISED, DATE_SHIPPED, 'HH24' )
```

| DATE_PROMISED | DATE_SHIPPED | RETURN VALUE |
|---|---|---|
| Jan 1 1997 12:00:00AM | Mar 29 1997 12:00:00PM | -2100 |
| Mar 29 1997 12:00:00PM | Jan 1 1997 12:00:00AM | 2100 |
| NULL | Dec 10 1997 5:55:10PM | NULL |
| Dec 10 1997 5:55:10PM | NULL | NULL |
| Jun 3 1997 1:13:46PM | Aug 23 1996 4:20:16PM | 6812.89166666667 |
| Feb 19 2004 12:00:00PM | Feb 19 2005 12:00:00PM | -8784 |

The following expressions return the number of days between the DATE_PROMISED and the DATE_SHIPPED columns:

```
DATE_DIFF( DATE_PROMISED, DATE_SHIPPED, 'D' )
DATE_DIFF( DATE_PROMISED, DATE_SHIPPED, 'DD' )
DATE_DIFF( DATE_PROMISED, DATE_SHIPPED, 'DDD' )
DATE_DIFF( DATE_PROMISED, DATE_SHIPPED, 'DY' )
DATE_DIFF( DATE_PROMISED, DATE_SHIPPED, 'DAY' )
```

| DATE_PROMISED | DATE_SHIPPED | RETURN VALUE |
|---|---|---|
| Jan 1 1997 12:00:00AM | Mar 29 1997 12:00:00PM | -87.5 |
| Mar 29 1997 12:00:00PM | Jan 1 1997 12:00:00AM | 87.5 |
| NULL | Dec 10 1997 5:55:10PM | NULL |
| Dec 10 1997 5:55:10PM | NULL | NULL |
| Jun 3 1997 1:13:46PM | Aug 23 1996 4:20:16PM | 283.870486111111 |
| Feb 19 2004 12:00:00PM | Feb 19 2005 12:00:00PM | -366 |

The following expressions return the number of months between the DATE_PROMISED and DATE_SHIPPED columns:

```
DATE_DIFF( DATE_PROMISED, DATE_SHIPPED, 'MM' )
DATE_DIFF( DATE_PROMISED, DATE_SHIPPED, 'MON' )
DATE_DIFF( DATE_PROMISED, DATE_SHIPPED, 'MONTH' )
```

| DATE_PROMISED | DATE_SHIPPED | RETURN VALUE |
|---|---|---|
| Jan 1 1997 12:00:00AM | Mar 29 1997 12:00:00PM | -2.91935483870968 |
| Mar 29 1997 12:00:00PM | Jan 1 1997 12:00:00AM | 2.91935483870968 |
| NULL | Dec 10 1997 5:55:10PM | NULL |
| Dec 10 1997 5:55:10PM | NULL | NULL |
| Jun 3 1997 1:13:46PM | Aug 23 1996 4:20:16PM | 9.3290162037037 |
| Feb 19 2004 12:00:00PM | Feb 19 2005 12:00:00PM | -12 |

The following expressions return the number of years between the DATE_PROMISED and DATE_SHIPPED columns:

```
DATE_DIFF( DATE_PROMISED, DATE_SHIPPED, 'Y' )
DATE_DIFF( DATE_PROMISED, DATE_SHIPPED, 'YY' )
DATE_DIFF( DATE_PROMISED, DATE_SHIPPED, 'YYY' )
DATE_DIFF( DATE_PROMISED, DATE_SHIPPED, 'YYYY' )
```

| DATE_PROMISED | DATE_SHIPPED | RETURN VALUE |
|---|---|---|
| Jan 1 1997 12:00:00AM | Mar 29 1997 12:00:00PM | -0.24327956989247 |
| Mar 29 1997 12:00:00PM | Jan 1 1997 12:00:00AM | 0.24327956989247 |
| NULL | Dec 10 1997 5:55:10PM | NULL |
| Dec 10 1997 5:55:10PM | NULL | NULL |
| Jun 3 1997 1:13:46PM | Aug 23 1996 4:20:16PM | 0.77741801697531 |
| Feb 19 2004 12:00:00PM | Feb 19 2005 12:00:00PM | -1 |

# DEC_BASE64

Decodes a base 64 encoded value and returns a string with the binary data representation of the data.

## Syntax

```
DEC_BASE64( value )
```

| Argument | Required/ Optional | Description |
|---|---|---|
| *value* | Required | String datatype. Data that you want to decode. |

### Return Value

Binary decoded value.

NULL if the input is a null value.

### Example

You encoded MQSeries message IDs and wrote them to a flat file. You want to read data from the flat file source, including the MQSeries message IDs. You can use DEC_BASE64 to decode the IDs and convert them to their original binary value.

# DECODE

Searches a column for a value that you specify. If the function finds the value, it returns a result value, which you define. You can build an unlimited number of searches within a DECODE function.

If you use DECODE to search for a value in a string column, you can either trim trailing blanks with the RTRIM function or include the blanks in the search string.

### Syntax

```
DECODE( value, first_search, first_result [, second_search, second_result]...[,default] )
```

| Argument | Required/ Optional | Description |
|---|---|---|
| *value* | Required | Any datatype except Binary. Passes the values you want to search. You can enter any valid expression. |
| *search* | Required | Any value with the same datatype as the value argument. Passes the values for which you want to search. The search value must match the value argument. You cannot search for a portion of a value. Also, the search value is case sensitive.<br>For example, if you want to search for the string 'Halogen Flashlight' in a particular column, you must enter 'Halogen Flashlight, not just 'Halogen'. If you enter 'Halogen', the search does not find a matching value. You can enter any valid expression. |
| *result* | Required | Any datatype except Binary. The value you want to return if the search finds a matching value. You can enter any valid expression. |
| *default* | Optional | Any datatype except Binary. The value you want to return if the search does not find a matching value. You can enter any valid expression. |

### Return Value

*First_result* if the search finds a matching value.

Default value if the search does not find a matching value.

NULL if you omit the default argument and the search does not find a matching value.

Even if multiple conditions are met, Data Integration returns the first matching result.

If the data contains multibyte characters and the DECODE expression compares string data, the return value depends on the code page of the Secure Agent that runs the task.

## DECODE and Datatypes

When you use DECODE, the datatype of the return value is always the same as the datatype of the result with the greatest precision.

For example, you have the following expression:

```
DECODE ( CONST_NAME,
         'Five', 5,
         'Pythagoras', 1.414213562,
         'Archimedes', 3.141592654,
         'Pi', 3.141592654 )
```

The return values in this expression are 5, 1.414213562, and 3.141592654. The first result is an Integer, and the other results are Decimal. The Decimal datatype has greater precision than Integer. This expression always writes the result as a Decimal.

If at least one result is Double, the datatype of the return value is Double.

You cannot create a DECODE function with both string and numeric return values.

For example, the following expression is invalid because the return values include both string and numeric values:

```
DECODE ( CONST_NAME,
         'Five', 5,
         'Pythagoras', '1.414213562',
         'Archimedes', '3.141592654',
         'Pi', 3.141592654 )
```

## Example

You might use DECODE in an expression that searches for a particular ITEM_ID and returns the ITEM_NAME:

```
DECODE( ITEM_ID, 10, 'Flashlight',
                 14, 'Regulator',
                 20, 'Knife',
                 40, 'Tank',
                 'NONE' )
```

| ITEM_ID | RETURN VALUE |
|---|---|
| 10 | Flashlight |
| 14 | Regulator |
| 17 | NONE |
| 20 | Knife |
| 25 | NONE |
| NULL | NONE |
| 40 | Tank |

DECODE returns the default value of NONE for items 17 and 25 because the search values did not match the ITEM_ID. Also, DECODE returns NONE for the NULL ITEM_ID.

The following expression tests multiple columns and conditions, evaluated in a top to bottom order for TRUE or FALSE:

```
DECODE( TRUE,
        Var1 = 22, 'Variable 1 was 22!',
        Var2 = 49, 'Variable 2 was 49!',
        Var1 < 23, 'Variable 1 was less than 23.',
```

```
            Var2 > 30, 'Variable 2 was more than 30.',
            'Variables were out of desired ranges.')
```

| Var1 | Var2 | RETURN VALUE |
|---|---|---|
| 21 | 47 | Variable 1 was less than 23. |
| 22 | 49 | Variable 1 was 22! |
| 23 | 49 | Variable 2 was 49! |
| 24 | 27 | Variables were out of desired ranges. |
| 25 | 50 | Variable 2 was more than 30. |

# DECOMPRESS

Decompresses data using the zlib compression algorithm. The zlib compression algorithm is compatible with WinZip. Use the DECOMPRESS function when you receive data over a wide area network.

## Syntax

```
DECOMPRESS( value, precision)
```

| Argument | Required/ Optional | Description |
|---|---|---|
| *value* | Required | Binary datatype. Data that you want to decompress. |
| precision | Optional | Integer datatype. |

## Return Value

Decompressed binary value of the input value.

NULL if the input is a null value.

## Example

Your organization has an online order service. You received compressed customer order data over a wide area network. You want to read the data using Data Integration and load the data to a data warehouse. You can decompress each row of data using DECOMPRESS for the row. Data Integration can then load the decompressed data to the target.

# ENC_BASE64

Encodes data by converting binary data to string data using Multipurpose Internet Mail Extensions (MIME) encoding.

Encode data when you want to store data in a database or file that does not allow binary data. You can also encode data to pass binary data in string format. The encoded data is approximately 33% longer than the original data. It displays as a set of random characters.

## Syntax

```
ENC_BASE64( value )
```

| Argument | Required/ Optional | Description |
|---|---|---|
| value | Required | Binary or String datatype. Data that you want to encode. |

## Return Value

Encoded value.

NULL if the input is a null value.

## Example

You want to read messages from an MQSeries source and write the data to a flat file target. You want to include the MQSeries message ID as part of the target data. However, the MsgID field is Binary, and the flat file target does not support binary data. Use ENC_BASE64 to encode the MsgID before Data Integration writes the data to the target.

# ERROR

Causes Data Integration to skip a row and issue an error message, which you define. Data Integration writes the skipped row and the error message into the error rows file.

Use ERROR in expressions to validate data. Generally, you use ERROR within an IIF or DECODE function to set rules for skipping rows. You might use ERROR to keep null values from passing into a target.

You can also include ERROR in expression to handle transformation errors.

## Syntax

```
ERROR( string )
```

| Argument | Required/ Optional | Description |
|---|---|---|
| *string* | Required | String value. The message you want to display when Data Integration skips a row based on the expression containing the ERROR function. The string can be any length. |

## Return Value

String.

## Example

The following example shows how to reference a mapping that calculates the average salary for employees in all departments of the organization, but skip negative values. The following expression nests the ERROR function in an IIF expression so that if Data Integration finds a negative salary in the Salary column, it skips the row and displays an error:

```
IIF( SALARY < 0, ERROR ('Error. Negative salary found. Row skipped.', EMP_SALARY )
```

| SALARY | RETURN VALUE |
|---|---|
| 10000 | 10000 |
| -15000 | 'Error. Negative salary found. Row skipped.' |
| NULL | NULL |
| 150000 | 150000 |
| 1005 | 1005 |

# EXP

Returns e raised to the specified power (exponent), where e=2.71828183. For example, EXP(2) returns 7.38905609893065. Use this function to analyze scientific and technical data. EXP is the reciprocal of the LN function, which returns the natural logarithm of a numeric value.

## Syntax

```
EXP( exponent )
```

| Argument | Required/ Optional | Description |
|---|---|---|
| *exponent* | Required | Numeric datatype. The value to which you want to raise e. The exponent in the equation e^*value*. You can enter any valid expression. |

## Return Value

Double value.

NULL if a value passed as an argument to the function is NULL.

## Example

The following expression uses the values stored in the Numbers column as the exponent value:

```
EXP( NUMBERS )
```

| NUMBERS | RETURN VALUE |
|---|---|
| 10 | 22026.4657948067 |
| -2 | 0.135335283236613 |
| 8.55 | 5166.754427176 |

| NUMBERS | RETURN VALUE |
|---|---|
| NULL | NULL |

# FIRST

Returns the first value found within a field or group. Optionally, you can apply a filter to limit the rows read. You can nest only one other aggregate function within FIRST.

Use only in mapping tasks.

## Syntax

```
FIRST( value [, filter_condition ] )
```

| Argument | Required/ Optional | Description |
|---|---|---|
| *value* | Required | Any data type except binary. Passes the values for which you want to return the first value. You can enter any valid transformation expression. |
| *filter_condition* | Optional | Limits the rows in the search. The filter condition must be a numeric value or evaluate to TRUE, FALSE, or NULL. You can enter any valid transformation expression. |

## Return Value

First value in a group.

NULL if all values passed to the function are NULL or if no rows are selected (for example, the filter condition evaluates to FALSE or NULL for all rows).

## Nulls

If a value is NULL, FIRST ignores the row. However, if all values passed from the field are NULL, FIRST returns NULL.

## Group By

FIRST groups values based on group by fields you define in the transformation, returning one result for each group.

If there is no group by field, FIRST treats all rows as one group, returning one value.

## Examples

The following expression returns the first value in the ITEM_NAME field with a price greater than $10.00:

```
FIRST( ITEM_NAME, ITEM_PRICE > 10 )
```

| ITEM_NAME | ITEM_PRICE |
|---|---|
| Flashlight | 35.00 |
| Navigation Compass | 8.05 |
| Regulator System | 150.00 |

| ITEM_NAME | ITEM_PRICE |
|---|---|
| Flashlight | 29.00 |
| Depth/Pressure Gauge | 88.00 |
| Flashlight | 31.00 |

**RETURN VALUE:** Flashlight

The following expression returns the first value in the ITEM_NAME field with a price greater than $40.00:

```
FIRST( ITEM_NAME, ITEM_PRICE > 40 )
```

| ITEM_NAME | ITEM_PRICE |
|---|---|
| Flashlight | 35.00 |
| Navigation Compass | 8.05 |
| Regulator System | 150.00 |
| Flashlight | 29.00 |
| Depth/Pressure Gauge | 88.00 |
| Flashlight | 31.00 |

**RETURN VALUE:** Regulator System

# FLOOR

Returns the largest integer less than or equal to the numeric value you pass to this function. For example, if you pass 3.14 to FLOOR, the function returns 3. If you pass 3.98 to FLOOR, the function returns 3. Likewise, if you pass -3.17 to FLOOR, the function returns -4.

## Syntax

```
FLOOR( numeric_value )
```

| Argument | Required/ Optional | Description |
|---|---|---|
| *numeric_value* | Required | Numeric datatype. You can enter any valid expression as long as it evaluates to numeric data. |

## Return Value

Integer if you pass a numeric value with declared precision between 0 and 28.

Double if you pass a numeric value with declared precision greater than 28.

NULL if a value passed to the function is NULL.

### Example

The following expression returns the largest integer less than or equal to the values in the PRICE column:

```
FLOOR( PRICE )
```

| PRICE | RETURN VALUE |
|---|---|
| 39.79 | 39 |
| 125.12 | 125 |
| 74.24 | 74 |
| NULL | NULL |
| -100.99 | -101 |

### Tip

You can perform arithmetic on the values you pass to FLOOR. For example, to multiply a numeric value by 10 and then calculate the largest integer that is less than the product, you might write the function as follows:

```
FLOOR( UNIT_PRICE * 10 )
```

# FV

Returns the future value of an investment, where you make periodic, constant payments and the investment earns a constant interest rate.

### Syntax

```
FV( rate, terms, payment [, present value, type] )
```

| Argument | Required/ Optional | Description |
|---|---|---|
| rate | Required | Numeric. Interest rate earned in each period. Expressed as a decimal number. Divide the percent rate by 100 to express it as a decimal number. Must be greater than or equal to 0. |
| terms | Required | Numeric. Number of periods or payments. Must be greater than 0. |
| payment | Required | Numeric. Payment amount due per period. Must be a negative number. |
| present value | Optional | Numeric. Current value of the investment. If you omit this argument, FV uses 0. |
| type | Optional | Integer. Timing of the payment. Enter 1 if payment is at the beginning of period. Enter 0 if payment is at the end of period. Default is 0. If you enter a value other than 0 or 1, Data Integration treats the value as 1. |

### Return Value

Numeric.

### Example

You deposit $2,000 into an account that earns 9% annual interest compounded monthly (monthly interest of 9%/12, or 0.75%). You plan to deposit $250 at the beginning of every month for the next 12 months. The following expression returns $5,337.96 as the account balance at the end of 12 months:

```
FV(0.0075, 12, -250, -2000, TRUE)
```

### Notes

To calculate interest rate earned in each period, divide the annual rate by the number of payments made in a year. The payment value and present value are negative because these are amounts that you pay.

# GET_DATE_PART

Returns the specified part of a date as an integer value. For example, if you create an expression that returns the month portion of the date, and pass a date such as Apr 1 1997 00:00:00, GET_DATE_PART returns 4.

### Syntax

```
GET_DATE_PART( date, format )
```

| Argument | Required/ Optional | Description |
|---|---|---|
| *date* | Required | Date/Time datatype. You can enter any valid expression. |
| *format* | Required | A format string specifying the portion of the date value you want to change. Enclose format strings within single quotation marks, for example, 'mm'. The format string is not case sensitive. Each format string returns the entire part of the date based on the default format of MM/DD/YYYY HH24:MI:SS.<br>For example, if you pass the date Apr 1 1997 to GET_DATE_PART, the format strings 'Y', 'YY', 'YYY', or 'YYYY' all return 1997. |

### Return Value

Integer representing the specified part of the date.

NULL if a value passed to the function is NULL.

### Example

The following expressions return the hour for each date in the DATE_SHIPPED column. 12:00:00AM returns 0 because the default date format is based on the 24 hour interval:

```
GET_DATE_PART( DATE_SHIPPED, 'HH' )
GET_DATE_PART( DATE_SHIPPED, 'HH12' )
GET_DATE_PART( DATE_SHIPPED, 'HH24' )
```

| DATE_SHIPPED | RETURN VALUE |
|---|---|
| Mar 13 1997 12:00:00AM | 0 |
| Sep 2 1997 2:00:01AM | 2 |
| Aug 22 1997 12:00:00PM | 12 |
| June 3 1997 11:30:44PM | 23 |

| DATE_SHIPPED | RETURN VALUE |
|---|---|
| NULL | NULL |

The following expressions return the day for each date in the DATE_SHIPPED column:

```
GET_DATE_PART( DATE_SHIPPED, 'D' )
GET_DATE_PART( DATE_SHIPPED, 'DD' )
GET_DATE_PART( DATE_SHIPPED, 'DDD' )
GET_DATE_PART( DATE_SHIPPED, 'DY' )
GET_DATE_PART( DATE_SHIPPED, 'DAY' )
```

| DATE_SHIPPED | RETURN VALUE |
|---|---|
| Mar 13 1997 12:00:00AM | 13 |
| June 3 1997 11:30:44PM | 3 |
| Aug 22 1997 12:00:00PM | 22 |
| NULL | NULL |

The following expressions return the month for each date in the DATE_SHIPPED column:

```
GET_DATE_PART( DATE_SHIPPED, 'MM' )
GET_DATE_PART( DATE_SHIPPED, 'MON' )
GET_DATE_PART( DATE_SHIPPED, 'MONTH' )
```

| DATE_SHIPPED | RETURN VALUE |
|---|---|
| Mar 13 1997 12:00:00AM | 3 |
| June 3 1997 11:30:44PM | 6 |
| NULL | NULL |

The following expression return the year for each date in the DATE_SHIPPED column:

```
GET_DATE_PART( DATE_SHIPPED, 'Y' )
GET_DATE_PART( DATE_SHIPPED, 'YY' )
GET_DATE_PART( DATE_SHIPPED, 'YYY' )
GET_DATE_PART( DATE_SHIPPED, 'YYYY' )
```

| DATE_SHIPPED | RETURN VALUE |
|---|---|
| Mar 13 1997 12:00:00AM | 1997 |
| June 3 1997 11:30:44PM | 1997 |
| NULL | NULL |

# GREATEST

Returns the greatest value from a list of input values. Use this function to return the greatest string, date, or number. By default, the match is case sensitive.

## Syntax

```
GREATEST( value1, [value2, ..., valueN,] CaseFlag )
```

| Argument | Required/ Optional | Description |
|---|---|---|
| value | Required | Any datatype except Binary. Datatype must be compatible with other values. Value you want to compare against other values. You must enter at least one value argument.<br>If the value is numeric, and other input values are numeric, all values use the highest precision possible. For example, if some values are Integer datatype and others are Double datatype, Data Integration converts the values to Double. |
| CaseFlag | Optional | Must be an integer. Determines whether the arguments in this function are case sensitive. You can enter any valid expression.<br>When *CaseFlag* is a number other than 0, the function is case sensitive.<br>When *CaseFlag* is a null value or 0, the function is not case sensitive. |

## Return Value

*value1* if it is the greatest of the input values, *value2* if it is the greatest of the input values, and so on.

NULL if all the arguments are null.

## Example

The following expression returns the greatest quantity of items ordered:

```
GREATEST( QUANTITY1, QUANTITY2, QUANTITY3 )
```

| QUANTITIY1 | QUANTITY2 | QUANTITY3 | RETURN VALUE |
|---|---|---|---|
| 150 | 756 | 27 | 756 |
| | | | NULL |
| 5000 | 97 | 17 | 5000 |
| 120 | 1724 | 965 | 1724 |

# IIF

Returns one of two values you specify, based on the results of a condition.

## Syntax

```
IIF( condition, value1 [,value2] )
```

| Argument | Required/ Optional | Description |
|---|---|---|
| *condition* | Required | The condition you want to evaluate. You can enter any valid expression that evaluates to TRUE or FALSE. |
| *value1* | Required | Any datatype except Binary. The value you want to return if the condition is TRUE. The return value is always the datatype specified by this argument. You can enter any valid expression, including another IIF expression. |
| *value2* | Optional | Any datatype except Binary. The value you want to return if the condition is FALSE. You can enter any valid expression, including another IIF expression. |

Unlike conditional functions in some systems, the FALSE (*value2*) condition in the IIF function is not required. If you omit *value2*, the function returns the following when the condition is FALSE:

- 0 if *value1* is a Numeric datatype.
- Empty string if *value1* is a String datatype.
- NULL if *value1* is a Date/Time datatype.

For example, the following expression does not include a FALSE condition and *value1* is a string datatype so Data Integration returns an empty string for each row that evaluates to FALSE:

```
IIF( SALES > 100, EMP_NAME )
```

| SALES | EMP_NAME | RETURN VALUE |
|---|---|---|
| 150 | John Smith | John Smith |
| 50 | Pierre Bleu | '' *(empty string)* |
| 120 | Sally Green | Sally Green |
| NULL | Greg Jones | '' *(empty string)* |

## Return Value

*value1* if the condition is TRUE.

*value2* if the condition is FALSE.

For example, the following expression includes the FALSE condition NULL so Data Integration returns NULL for each row that evaluates to FALSE:

```
IIF( SALES > 100, EMP_NAME, NULL )
```

| SALES | EMP_NAME | RETURN VALUE |
|---|---|---|
| 150 | John Smith | John Smith |

| SALES | EMP_NAME | RETURN VALUE |
|---|---|---|
| 50 | Pierre Bleu | NULL |
| 120 | Sally Green | Sally Green |
| NULL | Greg Jones | NULL |

If the data contains multibyte characters and the condition argument compares string data, the return value depends on the code page of the Secure Agent that runs the task.

## IIF and Datatypes

When you use IIF, the datatype of the return value is the same as the datatype of the result with the greatest precision.

For example, you have the following expression:

```
IIF( SALES < 100, 1, .3333 )
```

The TRUE result (1) is an integer and the FALSE result (.3333) is a decimal. The Decimal datatype has greater precision than Integer, so the datatype of the return value is always a Decimal.

When at least one result is Double, the datatype of the return value is Double.

## Special Uses of IIF

Use nested IIF statements to test multiple conditions. The following example tests for various conditions and returns 0 if sales is 0 or negative:

```
IIF( SALES > 0, IIF( SALES < 50, SALARY1, IIF( SALES < 100, SALARY2, IIF( SALES < 200,
SALARY3, BONUS))), 0 )
```

You can make this logic more readable by adding comments:

```
IIF( SALES > 0,
   --then test to see if sales is between 1 and 49:
      IIF( SALES < 50,
         --then return SALARY1
            SALARY1,
         --else test to see if sales is between 50 and 99:
            IIF( SALES < 100,
               --then return
                  SALARY2,
               --else test to see if sales is between 100 and 199:
                  IIF( SALES < 200,
                     --then return
                        SALARY3,
                     --else for sales over 199, return
                        BONUS)
                  )
           ),
   --else for sales less than or equal to zero, return
      0)
```

Use IIF in update strategies. For example:

```
IIF( ISNULL( ITEM_NAME ), DD_REJECT, DD_INSERT)
```

## Alternative to IIF

Use "DECODE" on page 97 instead of IIF in many cases. DECODE may improve readability. The following shows how you use DECODE instead of IIF using the first example from the previous section:

```
DECODE( TRUE,
   SALES > 0 and SALES < 50, SALARY1,
   SALES > 49 AND SALES < 100, SALARY2,
   SALES > 99 AND SALES < 200, SALARY3,
   SALES > 199, BONUS)
```

# IN

Matches input data to a list of values. By default, the match is case sensitive.

## Syntax

```
IN( valueToSearch, value1, [value2, ..., valueN,] CaseFlag )
```

| Argument | Required/ Optional | Description |
|---|---|---|
| valueToSearch | Required | Can be a string, date, or numeric value. Input value you want to match against a comma-separated list of values. |
| value | Required | Can be a string, date, or numeric value. Comma-separated list of values you want to search for. Values can be columns. There is no maximum number of values you can list. |
| CaseFlag | Optional | Must be an integer. Determines whether the arguments in this function are case sensitive. You can enter any valid expression.<br>When *CaseFlag* is a number other than 0, the function is case sensitive.<br>When *CaseFlag* is a null value or 0, the function is not case sensitive. |

## Return Value

TRUE (1) if the input value matches the list of values.

FALSE (0) if the input value does not match the list of values.

NULL if the input is a null value.

## Example

The following expression determines if the input value is a safety knife, chisel point knife, or medium titanium knife. The input values do not have to match the case of the values in the comma-separated list:

```
IN( ITEM_NAME, 'Chisel Point Knife', 'Medium Titanium Knife', 'Safety Knife', 0 )
```

| ITEM_NAME | RETURN VALUE |
|---|---|
| Stabilizing Vest | 0 (FALSE) |
| Safety knife | 1 (TRUE) |
| Medium Titanium knife | 1 (TRUE) |
| | NULL |

# INDEXOF

Finds the index of a value among a list of values. By default, the match is case sensitive.

### Syntax

```
INDEXOF( valueToSearch, string1, [string2, ..., stringN,] CaseFlag )
```

| Argument | Required/ Optional | Description |
|---|---|---|
| valueToSearch | Required | String datatype. Value you want to search for in the list of strings. |
| string | Required | String datatype. Comma-separated list of values you want to search against. Values can be in string format. There is no maximum number of values you can list. The value is case sensitive, unless you set MatchCase to 0. |
| CaseFlag | Required | Must be an integer. Determines whether the arguments in this function are case sensitive. You can enter any valid expression.<br>When *CaseFlag* is a number other than 0, the function is case sensitive.<br>When *CaseFlag* is a null value or 0, the function is not case sensitive. |

### Return Value

1 if the input value matches *string1*, 2 if the input value matches *string2*, and so on.

0 if the input value is not found.

NULL if the input is a null value.

### Example

The following expression determines if values from the ITEM_NAME column match the first, second, or third string:

```
INDEXOF( ITEM_NAME, 'diving hood', 'flashlight', 'safety knife')
```

| ITEM_NAME | RETURN VALUE |
|---|---|
| Safety Knife | 0 |
| diving hood | 1 |
| Compass | 0 |
| safety knife | 3 |
| flashlight | 2 |

Safety Knife returns a value of 0 because it does not match the case of the input value.

# INITCAP

Capitalizes the first letter in each word of a string and converts all other letters to lowercase. Words are delimited by white space (a blank space, formfeed, newline, carriage return, tab, or vertical tab) and

characters that are not alphanumeric. For example, if you pass the string '…THOMAS', the function returns Thomas.

## Syntax

```
INITCAP( string )
```

| Argument | Required/ Optional | Description |
|---|---|---|
| *string* | Required | Any datatype except Binary. You can enter any valid expression. |

## Return Value

String. If the data contains multibyte characters, the return value depends on the code page of the Secure Agent that runs the task.

NULL if a value passed to the function is NULL.

## Example

The following expression capitalizes all names in the FIRST_NAME column:

```
INITCAP( FIRST_NAME )
```

| FIRST_NAME | RETURN VALUE |
|---|---|
| ramona | Ramona |
| 18-albert | 18-Albert |
| NULL | NULL |
| ?!SAM | ?!Sam |
| THOMAS | Thomas |
| PierRe | Pierre |

# INSTR

Returns the position of a character set in a string, counting from left to right.

## Syntax

```
INSTR( string, search_value [,start [,occurrence]] )
```

| Argument | Required/ Optional | Description |
|---|---|---|
| *string* | Required | The string must be a character string. Passes the value you want to evaluate. You can enter any valid expression. The results of the expression must be a character string. If not, INSTR converts the value to a string before evaluating it. |
| *search_value* | Required | Any value. The search value is case sensitive. The set of characters you want to search for. The search_value must match a part of the string. For example, if you write `INSTR('Alfred Pope', 'Alfred Smith')` the function returns 0.<br>You can enter any valid expression. If you want to search for a character string, enclose the characters you want to search for in single quotation marks, for example 'abc'. |
| *start* | Optional | Must be an integer value. The position in the string where you want to start the search. You can enter any valid expression.<br>The default is 1, meaning that INSTR starts the search at the first character in the string.<br>If the start position is 0, INSTR searches from the first character in the string. If the start position is a positive number, INSTR locates the start position by counting from the beginning of the string. If the start position is a negative number, INSTR locates the start position by counting from the end of the string. If you omit this argument, the function uses the default value of 1. |
| *occurrence* | Optional | A positive integer greater than 0. You can enter any valid expression. If the search value appears more than once in the string, you can specify which occurrence you want to search for. For example, you would enter 2 to search for the second occurrence from the start position.<br>If you omit this argument, the function uses the default value of 1, meaning that INSTR searches for the first occurrence of the search value. If you pass a decimal, Data Integration rounds it to the nearest integer value. If you pass a negative integer or 0, the mapping fails when you run a workflow. |

## Return Value

Integer if the search is successful. Integer represents the position of the first character in the *search_value*, counting from left to right.

0 if the search is unsuccessful.

NULL if a value passed to the function is NULL.

## Example

The following expression returns the position of the first occurrence of the letter ‘a’, starting at the beginning of each company name. Because the *search_value* argument is case sensitive, it skips the ‘A’ in ‘Blue Fin Aqua Center’, and returns the position for the ‘a’ in ‘Aqua’:

```
INSTR( COMPANY, 'a' )
```

| COMPANY | RETURN VALUE |
|---|---|
| Blue Fin Aqua Center | 13 |

| COMPANY | RETURN VALUE |
|---|---|
| Maco Shark Shop | 2 |
| Scuba Gear | 5 |
| Frank's Dive Shop | 3 |
| VIP Diving Club | 0 |

The following expression returns the position of the second occurrence of the letter ‘a’, starting at the beginning of each company name. Because the *search_value* argument is case sensitive, it skips the ‘A’ in ‘Blue Fin Aqua Center’, and returns 0:

```
INSTR( COMPANY, 'a', 1, 2 )
```

| COMPANY | RETURN VALUE |
|---|---|
| Blue Fin Aqua Center | 0 |
| Maco Shark Shop | 8 |
| Scuba Gear | 9 |
| Frank's Dive Shop | 0 |
| VIP Diving Club | 0 |

The following expression returns the position of the second occurrence of the letter ‘a’ in each company name, starting from the last character in the company name. Because the *search_value* argument is case sensitive, it skips the ‘A’ in 'Blue Fin Aqua Center’, and returns 0:

```
INSTR( COMPANY, 'a', -1, 2 )
```

| COMPANY | RETURN VALUE |
|---|---|
| Blue Fin Aqua Center | 0 |
| Maco Shark Shop | 2 |
| Scuba Gear | 5 |
| Frank's Dive Shop | 0 |
| VIP Diving Club | 0 |

The following expression returns the position of the first character in the string ‘Blue Fin Aqua Center’ (starting from the last character in the company name):

```
INSTR( COMPANY, 'Blue Fin Aqua Center', -1, 1 )
```

| COMPANY | RETURN VALUE |
|---|---|
| Blue Fin Aqua Center | 1 |
| Maco Shark Shop | 0 |
| Scuba Gear | 0 |

| COMPANY | RETURN VALUE |
| --- | --- |
| Frank's Dive Shop | 0 |
| VIP Diving Club | 0 |

### Using Nested INSTR

You can nest the INSTR function within other functions to accomplish more complex tasks.

The following expression evaluates a string, starting from the end of the string. The expression finds the last (rightmost) space in the string and then returns all characters to the left of it:

```
SUBSTR( CUST_NAME,1,INSTR( CUST_NAME,' ' ,-1,1 ))
```

| CUST_NAME | RETURN VALUE |
| --- | --- |
| PATRICIA JONES | PATRICIA |
| MARY ELLEN SHAH | MARY ELLEN |

The following expression removes the character '#' from a string:

```
SUBSTR( CUST_ID, 1, INSTR(CUST_ID, '#')-1 ) || SUBSTR( CUST_ID, INSTR(CUST_ID, '#')+1 )
```

| CUST_ID | RETURN VALUE |
| --- | --- |
| ID#33 | ID33 |
| #A3577 | A3577 |
| SS #712403399 | SS 712403399 |

# IS_DATE

Returns whether a string value is a valid date.

A valid date is any string in the default date format of MM/DD/YYYY HH24:MI:SS. If the strings you want to test are not in the default date format, use the TO_DATE format strings to specify the date format. If the strings passed to IS_DATE do not match the format string specified, the function returns FALSE (0). If the strings match the format string, the function returns TRUE (1).

IS_DATE evaluates strings and returns an integer value.

The target column for an IS_DATE expression must be String or Numeric datatype.

You might use IS_DATE to test or filter data in a flat file before writing it to a target.

Use the RR format string with IS_DATE instead of the YY format string. In most cases, the two format strings return the same values, but there are some unique cases where YY returns incorrect results. For example, the expression IS_DATE('02/29/00', 'YY') is internally computed as IS_DATE(02/29/1900 00:00:00), which returns false. However, Data Integration computes the expression IS_DATE('02/29/00', 'RR') as IS_DATE(02/29/2000 00:00:00), which returns TRUE. In the first case, year 1900 is not a leap year, so there is no February 29th.

**Note:** IS_DATE uses the same format strings as TO_DATE.

## Syntax

```
IS_DATE( value [,format] )
```

| Argument | Required/ Optional | Description |
|---|---|---|
| *value* | Required | Must be a string datatype. Passes the rows you want to evaluate. You can enter any valid expression. |
| *format* | Optional | Enter a valid TO_DATE format string. The format string must match the parts of the *string* argument. For example, if you pass the string 'Mar 15 1997 12:43:10AM', you must use the format string 'MON DD YYYY HH12:MI:SSAM'. If you omit the format string, the string value must be in the default date of MM/DD/YYYY HH24:MI:SS.<br>For more information about TO_DATE and IS_DATE format strings, see "TO_DATE and IS_DATE format strings" on page 37. |

## Return Value

TRUE (1) if the row is a valid date.

FALSE (0) if the row is not a valid date.

NULL if a value in the expression is NULL or if the format string is NULL.

**Warning:** The format of the IS_DATE string must match the format string, including any date separators. If it does not, Data Integration might return inaccurate values or skip the row.

## Example

The following expression checks the INVOICE_DATE column for valid dates:

```
IS_DATE( INVOICE_DATE )
```

This expression returns data similar to the following:

```
INVOICE_DATE                    RETURN VALUE

NULL                            NULL

'180'                           0 (FALSE)

'04/01/98'                      0 (FALSE)

'04/01/1998 00:12:15'           1 (TRUE)

'02/31/1998 12:13:55'           0 (FALSE)   (February does not have 31 days)

'John Smith'                    0 (FALSE)
```

The following IS_DATE expression specifies a format string of 'YYYY/MM/DD':

```
IS_DATE( INVOICE_DATE, 'YYYY/MM/DD' )
```

If the string value does not match this format, IS_DATE returns FALSE:

```
INVOICE_DATE                    RETURN VALUE

NULL                            NULL

'180'                           0 (FALSE)
```

| INVOICE_DATE | RETURN VALUE | |
|---|---|---|
| '04/01/98' | 0 (FALSE) | |
| '1998/01/12' | 1 (TRUE) | |
| '1998/11/21 00:00:13' | 0 (FALSE) | |
| '1998/02/31' | 0 (FALSE) | (*February does not have 31 days*) |
| 'John Smith' | 0 (FALSE) | |

The following example shows how you use IS_DATE to test data before using TO_DATE to convert the strings to dates. This expression checks the values in the INVOICE_DATE column and converts each valid date to a date value. If the value is not a valid date, Data Integration returns ERROR and skips the row.

This example returns a Date/Time value. Therefore, the target column for the expression needs to be Date/Time:

```
IIF( IS_DATE ( INVOICE_DATE, 'YYYY/MM/DD' ), TO_DATE( INVOICE_DATE ), ERROR('Not a valid date' ) )
```

| INVOICE_DATE | RETURN VALUE |
|---|---|
| NULL | NULL |
| '180' | 'Not a valid date' |
| '04/01/98' | 'Not a valid date' |
| '1998/01/12' | 1998/01/12 |
| '1998/11/21 00:00:13' | 'Not a valid date' |
| '1998/02/31' | 'Not a valid date' |
| 'John Smith' | 'Not a valid date' |

# IS_NUMBER

Returns whether a string is a valid number.

A valid number consists of the following parts:

- Optional space before the number
- Optional sign (+/-)
- One or more digits with an optional decimal point
- Optional scientific notation, such as the letter ‘e’ or ‘E’ (and the letter ‘d’ or ‘D’ on Windows) followed by an optional sign (+/-), followed by one or more digits
- Optional white space following the number

The following numbers are all valid:

```
' 100  '
'   +100'
```

```
'-100'
'-3.45e+32'
'+3.45E-32'
'+3.45d+32' (Windows only)
'+3.45D-32' (Windows only)
'.6804'
```

The target column for an IS_NUMBER expression must be a String or Numeric datatype.

You might use IS_NUMBER to test or filter data in a flat file before writing it to a target.

## Syntax

```
IS_NUMBER( value )
```

| Argument | Required/ Optional | Description |
|---|---|---|
| *value* | Required | Must be a String datatype. Passes the rows you want to evaluate. You can enter any valid expression. |

## Return Value

TRUE (1) if the row is a valid number.

FALSE (0) if the row is not a valid number.

NULL if a value in the expression is NULL.

## Example

The following expression checks the ITEM_PRICE column for valid numbers:

```
IS_NUMBER( ITEM_PRICE )
```

```
ITEM_PRICE          RETURN VALUE

'123.00'            1 (True)

'-3.45e+3'          1 (True)

'-3.45D-3'          1 (True - Windows only)

'-3.45d-3'          0 (False - UNIX only)

'3.45E-'            0 (False)  Incomplete number

'    '              0 (False)  Consists entirely of whitespace

''                  0 (False)  Empty string

'+123abc'           0 (False)

'  123'             1 (True)  Leading whitespace

'123  '             1 (True)  Trailing whitespace

'ABC'               0 (False)

'-ABC'              0 (False)

NULL                NULL
```

Use IS_NUMBER to test data before using one of the numeric conversion functions, such as TO_FLOAT. For example, the following expression checks the values in the ITEM_PRICE column and converts each valid number to a double-precision floating point value. If the value is not a valid number, Data Integration returns 0.00:

```
IIF( IS_NUMBER ( ITEM_PRICE ), TO_FLOAT( ITEM_PRICE ), 0.00 )
```

| ITEM_PRICE | RETURN VALUE | |
|---|---|---|
| '123.00' | 123 | |
| '-3.45e+3' | -3450 | |
| '3.45E-3' | 0.00345 | |
| '  ' | 0.00 | *Consists entirely of whitespace* |
| '' | 0.00 | *Empty string* |
| '+123abc' | 0.00 | |
| ''  123ABC' | 0.00 | |
| 'ABC' | 0.00 | |
| '-ABC' | 0.00 | |
| NULL | NULL | |

# IS_SPACES

Returns whether a string value consists entirely of spaces. A space is a blank space, a formfeed, a newline, a carriage return, a tab, or a vertical tab.

IS_SPACES evaluates an empty string as FALSE because there are no spaces. To test for an empty string, use LENGTH.

## Syntax

```
IS_SPACES( value )
```

| Argument | Required/ Optional | Description |
|---|---|---|
| *value* | Required | Must be a string datatype. Passes the rows you want to evaluate. You can enter any valid expression. |

## Return Value

TRUE (1) if the row consists entirely of spaces.

FALSE (0) if the row contains data.

NULL if a value in the expression is NULL.

### Example

The following expression checks the ITEM_NAME column for rows that consist entirely of spaces:

```
IS_SPACES( ITEM_NAME )
```

| ITEM_NAME | RETURN VALUE |
|---|---|
| Flashlight | 0 (False) |
| | 1 (True) |
| Regulator system | 0 (False) |
| NULL | NULL |
| '' | 0 (FALSE) (*Empty string does not contain spaces.*) |

### Tip

Use IS_SPACES to avoid writing spaces to a character column in a target table. For example, if you want to write customer names to a fixed length CHAR(5) column in a target table, you might want to write '00000' instead of spaces. You would create an expression similar to the following:

```
IIF( IS_SPACES( CUST_NAMES ), '00000', CUST_NAMES )
```

# ISNULL

Returns whether a value is NULL. ISNULL evaluates an empty string as FALSE.

**Note:** To test for empty strings, use LENGTH.

### Syntax

```
ISNULL( value )
```

| Argument | Required/ Optional | Description |
|---|---|---|
| *value* | Required | Any datatype except Binary. Passes the rows you want to evaluate. You can enter any valid expression. |

### Return Value

TRUE (1) if the value is NULL.

FALSE (0) if the value is not NULL.

### Example

The following example checks for null values in the items table:

```
ISNULL( ITEM_NAME )
```

| ITEM_NAME | RETURN VALUE |
|---|---|
| Flashlight | 0 (FALSE) |

```
ITEM_NAME                RETURN VALUE

NULL                     1 (TRUE)

Regulator system         0 (FALSE)

''                       0 (FALSE)  Empty string is not NULL
```

# LAG

In advanced mode, the LAG function returns data from a preceding row in an Expression transformation. Use this function to compare values in the current row with values in a preceding row.

To use the LAG function, you must configure partition and order keys as window properties and in the Expression transformation.

## Syntax

```
LAG ( column_name, offset, default )
```

The following table describes the arguments for this command:

| Argument | Required/ Optional | Description |
|---|---|---|
| *column_name* | Required | The target column or expression that the function operates on. |
| *offset* | Required | Integer data type. The number of rows before the current row from which to obtain a value. |
| *default* | Optional | The default value to return if the offset is outside the bounds of the partition or table. Default is NULL.<br>You can specify a default argument that is the same data type as the input value or goes with the offset argument.<br>You cannot specify a default argument that contains a complex data type or a SYSTIMESTAMP argument. |

## Return Value

The data type of the specified *column_name*.

*Default* if the return value is outside the bounds of the specified partition.

NULL if *default* is omitted or set to NULL.

## Example 1

The following expression returns the date that the previous order was placed:

```
LAG ( ORDER_DATE, 1, NULL )
```

The following table shows the order information for this command:

| ORDER_DATE | ORDER_ID | RETURN VALUE |
|---|---|---|
| 2017/09/25 | 1 | NULL |
| 2017/09/26 | 2 | 2017/09/25 |
| 2017/09/27 | 3 | 2017/09/26 |
| 2017/09/28 | 4 | 2017/09/27 |
| 2017/09/29 | 5 | 2017/09/28 |
| 2017/09/30 | 6 | 2017/09/29 |

The lag value of the first row is outside the partition, so the function returns the default value of NULL.

## Example 2

Your organization receives GPS pings from vehicles that include trip and event IDs and a time stamp. You want to calculate the time difference between each ping.

The following expression calculates the time difference between the current row and the previous row for two separate trips:

```
DATE_DIFF( EVENT_TIME, LAG ( EVENT_TIME, 1, NULL ), 'ss' )
```

You partition the data by trip ID and order by event ID.

The following tables shows trip information for this command:

| TRIP_ID | EVENT_ID | EVENT_TIME | RETURN VALUE |
|---|---|---|---|
| 101 | 1 | 2017-05-03 12:00:00 | NULL |
| 101 | 2 | 2017-05-03 12:00:34 | 34 |
| 101 | 3 | 2017-05-03 12:02:00 | 86 |
| 102 | 1 | 2017-05-03 12:00:00 | NULL |
| 102 | 2 | 2017-05-03 12:01:56 | 116 |
| 102 | 3 | 2017-05-03 12:02:00 | 4 |

The lag values of the first and fourth row are outside of the specified partition, so the function returns two default NULL values.

# LAST

Returns the last row in the selected field. Optionally, you can apply a filter to limit the rows read. You can nest only one other aggregate function within LAST.

Use only in mapping tasks.

## Syntax

```
LAST( value [, filter_condition ] )
```

| Argument | Required/ Optional | Description |
|---|---|---|
| *value* | Required | Any data type except binary. Passes the values for which you want to return the last row. You can enter any valid transformation expression. |
| *filter_condition* | Optional | Limits the rows in the search. The filter condition must be a numeric value or evaluate to TRUE, FALSE, or NULL. You can enter any valid transformation expression. |

## Return Value

Last row in a field.

NULL if all values passed to the function are NULL, or if no rows are selected (for example, the filter condition evaluates to FALSE or NULL for all rows).

## Example

The following expression returns the last row in the ITEMS_NAME field with a price greater than $10.00:

```
LAST( ITEM_NAME, ITEM_PRICE > 10 )
```

| ITEM_NAME | ITEM_PRICE |
|---|---|
| Flashlight | 35.00 |
| Navigation Compass | 8.05 |
| Regulator System | 150.00 |
| Flashlight | 29.00 |
| Depth/Pressure Gauge | 88.00 |
| Vest | 31.00 |

**RETURN VALUE:** Vest

# LAST_DAY

Returns the date of the last day of the month for each date in a column.

## Syntax

```
LAST_DAY( date )
```

| Argument | Required/ Optional | Description |
|---|---|---|
| *date* | Required | Date/Time datatype. Passes the dates for which you want to return the last day of the month. You can enter any valid expression that evaluates to a date. |

## Return Value

Date. The last day of the month for that date value you pass to this function.

NULL if a value in the selected column is NULL.

## Null

If a value is NULL, LAST_DAY ignores the row. However, if all values passed from the column are NULL, LAST_DAY returns NULL.

## Example

The following expression returns the last day of the month for each date in the ORDER_DATE column:

```
LAST_DAY( ORDER_DATE )
```

| ORDER_DATE | RETURN VALUE | |
|---|---|---|
| Apr 1 1998 12:00:00AM | Apr 30 1998 12:00:00AM | |
| Jan 6 1998 12:00:00AM | Jan 31 1998 12:00:00AM | |
| Feb 2 1996 12:00:00AM | Feb 29 1996 12:00:00AM | *(Leap year)* |
| NULL | NULL | |
| Jul 31 1998 12:00:00AM | Jul 31 1998 12:00:00AM | |

You can nest TO_DATE to convert string values to a date. TO_DATE always includes time information. If you pass a string that does not have a time value, the date returned will include the time 00:00:00.

The following example returns the last day of the month for each order date in the same format as the string:

```
LAST_DAY( TO_DATE( ORDER_DATE, 'DD-MON-YY' ))
```

| ORDER_DATE | RETURN VALUE |
|---|---|
| '18-NOV-98' | Nov 30 1998 00:00:00 |
| '28-APR-98' | Apr 30 1998 00:00:00 |
| NULL | NULL |
| '18-FEB-96' | Feb 29 1996 00:00:00 *(Leap year)* |

# LEAD

In advanced mode, the LEAD function returns data from the following row in an Expression transformation. Use this function to compare values in the current row with values in the following row.

To use the LEAD function, you must configure partition and order keys as window properties and in the Expression transformation.

## Syntax

```
LEAD ( column_name, offset, default )
```

The following table describes the arguments for this command:

| Argument | Required/ Optional | Description |
|---|---|---|
| *column_name* | Required | The target column or expression that the function operates on. |
| *offset* | Required | Integer data type. The number of rows after the current row from which to obtain a value. |
| *default* | Optional | The default value to be returned if the offset is outside the bounds of the partition or table. Default is NULL.<br>You can specify a default argument that is the same data type as the input value or goes with the offset argument.<br>You cannot specify a default argument that contains a complex data type or a SYSTIMESTAMP argument. |

## Return Value

The data type of the specified *column_name*.

*Default* if the return value is outside the bounds of the specified partition.

NULL if *default* is omitted or set to NULL.

## Example 1

This example shows employee names and hire dates data for use in the LEAD function.

The following table shows employee information:

| EMPLOYEE | HIRE_DATE | RETURN VALUE |
|---|---|---|
| Hynes | 2012/12/07 | 2014/05/18 |
| Williams | 2014/05/18 | 2015/07/24 |
| Pritchard | 2015/07/24 | 2015/12/24 |
| Snyder | 2015/12/24 | 2016/11/15 |
| Troy | 2016/11/15 | 2017/08/10 |
| Randolph | 2017/08/10 | NULL |

For each employee, the following expression returns the date that the next employee was hired:

```
LEAD ( HIRE_DATE, 1, NULL )
```

There is no lead value available for the last row, so the function returns the default value of NULL.

## Example 2

This example shows sales quota values for two calendar years to be used in the LEAD function.

You partition the data by year and order by quarter.

The following table shows sales quota data for use in the function:

| YEAR | QUARTER | SALES_QUOTA | QUOTA_DIFF |
|---|---|---|---|
| 2016 | 1* | 300 | 7700 |
| 2016 | 2* | 7000 | 0 |
| 2016 | 3 | 8000 | 0 |
| 2017 | 1 | 5000 | 4000 |
| 2017 | 2 | 400 | 0 |
| 2017 | 3 | 9000 | 0 |

**The lead values of the second and third quarter are outside the specified partition, so the function returns a value of "0."*

The following expression returns the difference in sales quota values between the first quarter to the third quarter of two calendar years:

```
LEAD ( Sales_Quota, 2, 0 ) - Sales_Quota
```

# LEAST

Returns the smallest value from a list of input values. By default, the match is case sensitive.

## Syntax

```
LEAST( value1, [value2, ..., valueN,] CaseFlag )
```

| Argument | Required/ Optional | Description |
|---|---|---|
| value | Required | Any datatype except Binary. Datatype must be compatible with other values. Value you want to compare against other values. You must enter at least one value argument.<br>If the value is Numeric, and other input values are of other numeric datatypes, all values use the highest precision possible. For example, if some values are of the Integer datatype and others are of the Double datatype, Data Integration converts the values to Double. |
| CaseFlag | Optional | Must be an integer. Determines whether the arguments in this function are case sensitive. You can enter any valid expression.<br>When *CaseFlag* is a number other than 0, the function is case sensitive.<br>When *CaseFlag* is a null value or 0, the function is not case sensitive. |

## Return Value

*value1* if it is the smallest of the input values, *value2* if it is the smallest of the input values, and so on.

NULL if all the arguments are null.

## Example

The following expression returns the smallest quantity of items ordered:

```
LEAST( QUANTITY1, QUANTITY2, QUANTITY3 )
```

| QUANTITIY1 | QUANTITY2 | QUANTITY3 | RETURN VALUE |
|---|---|---|---|
| 150 | 756 | 27 | 27 |
| | | | NULL |
| 5000 | 97 | 17 | 17 |
| 120 | 1724 | 965 | 120 |

# LENGTH

Returns the number of characters in a string, including trailing blanks.

## Syntax

```
LENGTH( string )
```

| Argument | Required/ Optional | Description |
|---|---|---|
| *string* | Required | String datatype. The strings you want to evaluate. You can enter any valid expression. |

## Return Value

Integer representing the length of the string.

NULL if a value passed to the function is NULL.

## Example

The following expression returns the length of each customer name:

```
LENGTH( CUSTOMER_NAME )
```

| CUSTOMER_NAME | RETURN VALUE |
|---|---|
| Bernice Davis | 13 |
| NULL | NULL |
| John Baer | 9 |
| Greg Brown | 10 |

## Tips

Use LENGTH to test for empty string conditions. If you want to find fields in which customer name is empty, use an expression such as:

```
IIF( LENGTH( CUSTOMER_NAME ) = 0, 'EMPTY STRING' )
```

To test for a null field, use ISNULL. To test for spaces, use IS_SPACES.

# LN

Returns the natural logarithm of a numeric value.

For example, `LN(3)` returns `1.098612`. You usually use this function to analyze scientific data rather than business data.

This function is the reciprocal of the function EXP.

### Syntax

```
LN( numeric_value )
```

| Argument | Required/ Optional | Description |
|---|---|---|
| *numeric_value* | Required | Numeric datatype. It must be a positive number, greater than 0. Passes the values for which you want to calculate the natural logarithm. You can enter any valid expression. |

### Return Value

Double value.

NULL if a value passed to the function is NULL.

### Example

The following expression returns the natural logarithm for all values in the NUMBERS column:

```
LN( NUMBERS )
```

| NUMBERS | RETURN VALUE |
|---|---|
| 10 | 2.302585092994 |
| 125 | 4.828313737302 |
| 0.96 | -0.04082199452026 |
| NULL | NULL |
| -90 | None. (Data Integration writes the row to the error rows file.) |
| 0 | None. (Data Integration writes the row to the error rows file.) |

**Note:** When you pass a negative number or 0, Data Integration writes the row into the error rows file. The *numeric_value* must be a positive number greater than 0.

# LOG

Returns the logarithm of a numeric value.

Most often, you use this function to analyze scientific data rather than business data.

## Syntax

```
LOG( base, exponent )
```

| Argument | Required/ Optional | Description |
|---|---|---|
| *base* | Required | The base of the logarithm. Must be a positive numeric value other than 0 or 1. Any valid expression that evaluates to a positive number other than 0 or 1. |
| *exponent* | Required | The exponent of the logarithm. Must be a positive numeric value greater than 0. Any valid expression that evaluates to a positive number greater than 0. |

## Return Value

Double value.

NULL if a value passed to the function is NULL.

## Example

The following expression returns the logarithm for all values in the NUMBERS column:

```
LOG( BASE, EXPONENT )
```

```
BASE   EXPONENT  RETURN VALUE

15     1         0

.09    10        -0.956244644696599

NULL   18        NULL

35.78  NULL      NULL

-9     18        None. (Data Integration writes the row to the error rows file.)

0      5         None. (Data Integration writes the row to the error rows file.)

10     -2        None. (Data Integration writes the row to the error rows file.)
```

If you pass a negative number, 0, or 1 as a base value, or if you pass a negative value for the exponent, Data Integration writes the row to the error rows file.

# LOWER

Converts uppercase string characters to lowercase.

## Syntax

```
LOWER( string )
```

| Argument | Required/ Optional | Description |
|---|---|---|
| *string* | Required | Any string value. The argument passes the string values that you want to return as lowercase. You can enter any valid expression that evaluates to a string. |

## Return Value

Lowercase character string. If the data contains multibyte characters, the return value depends on the code page of the Secure Agent that runs the task.

NULL if a value in the selected column is NULL.

## Example

The following expression returns all first names to lowercase:

```
LOWER( FIRST_NAME )
```

| FIRST_NAME | RETURN VALUE |
|---|---|
| antonia | antonia |
| NULL | NULL |
| THOMAS | thomas |
| PierRe | pierre |
| BERNICE | bernice |

# LPAD

Adds a set of blanks or characters to the beginning of a string to set the string to a specified length.

## Syntax

```
LPAD( first_string, length [,second_string] )
```

| Argument | Required/ Optional | Description |
|---|---|---|
| *first_string* | Required | Can be a character string. The strings you want to change. You can enter any valid expression. |
| *length* | Required | Must be a positive integer literal. This argument specifies the length you want each string to be. When *length* is a negative number, LPAD returns NULL. |
| second_string | Optional | Can be any string value. The characters you want to append to the left-side of the *first_string* values. You can enter any valid expression. You can enter a specific string literal. However, enclose the characters you want to add to the beginning of the string within single quotation marks, as in 'abc'. This argument is case sensitive. If you omit the *second_string*, the function pads the beginning of the first string with blanks. |

## Return Value

String of the specified length.

NULL if a value passed to the function is NULL or if *length* is a negative number.

## Example

The following expression standardizes numbers to six digits by padding them with leading zeros.

```
LPAD( PART_NUM, 6, '0')
```

| PART_NUM | RETURN VALUE |
|---|---|
| 702 | 000702 |
| 1 | 000001 |
| 0553 | 000553 |
| 484834 | 484834 |

LPAD counts the length from left to right. If the first string is longer than the length, LPAD truncates the string from right to left. For example, LPAD(‘alphabetical’, 5, ‘x’) returns the string ‘alpha’.

If the second string is longer than the total characters needed to return the specified length, LPAD uses a portion of the second string:

```
LPAD( ITEM_NAME, 16, '*..*' )
```

| ITEM_NAME | RETURN VALUE |
|---|---|
| Flashlight | *..**.Flashlight |
| Compass | *..**..**Compass |
| Regulator System | Regulator System |

| ITEM_NAME | RETURN VALUE |
|---|---|
| Safety Knife | *..*Safety Knife |

The following expression shows how LPAD handles negative values for the `length` argument for each row in the ITEM_NAME column:

```
LPAD( ITEM_NAME, -5, '.')
```

| ITEM_NAME | RETURN VALUE |
|---|---|
| Flashlight | NULL |
| Compass | NULL |
| Regulator System | NULL |

# LTRIM

Removes blanks or characters from the beginning of a string. You can use LTRIM with IIF or DECODE in an expression to avoid spaces in a target table.

If you do not specify a *trim_set* parameter in the expression, LTRIM removes only single-byte spaces.

If you use LTRIM to remove characters from a string, LTRIM compares the *trim_set* to each character in the *string* argument, character-by-character, starting with the left side of the string. If the character in the string matches any character in the *trim_set*, LTRIM removes it. LTRIM continues comparing and removing characters until it fails to find a matching character in the *trim_set*. Then it returns the string, which does not include matching characters.

## Syntax

```
LTRIM( string [, trim_set] )
```

| Arguments | Required/ Optional | Description |
|---|---|---|
| *string* | Required | Any string value. Passes the strings you want to modify. You can enter any valid expression. Use operators to perform comparisons or concatenate strings before removing characters from the beginning of a string. |
| *trim_set* | Optional | Any string value. Passes the characters you want to remove from the beginning of the first string. You can enter any valid expression. You can also enter a character string. However, you must enclose the characters you want to remove from the beginning of the string within single quotation marks, for example, 'abc'. If you omit the second string, the function removes any blanks from the beginning of the string.<br>LTRIM is case sensitive. For example, if you want to remove the 'A' character from the string 'Alfredo', you would enter 'A', not 'a'. |

## Return Value

String. The string values with the specified characters in the *trim_set* argument removed.

NULL if a value passed to the function is NULL. If the *trim_set* is NULL, the function returns NULL.

### Example

The following expression removes the characters ‘S’ and ‘.’ from the strings in the LAST_NAME column:

```
LTRIM( LAST_NAME, 'S.')
```

| LAST_NAME | RETURN VALUE |
|---|---|
| Nelson | Nelson |
| Osborne | Osborne |
| NULL | NULL |
| S. MacDonald | MacDonald |
| Sawyer | awyer |
| H. Bender | H. Bender |
| Steadman | teadman |

LTRIM removes ‘S.’ from S. MacDonald and the ‘S’ from both Sawyer and Steadman, but not the period from H. Bender. This is because LTRIM searches, character-by-character, for the set of characters you specify in the *trim_set* argument. If the first character in the string matches the first character in the *trim_set*, LTRIM removes it. Then LTRIM looks at the second character in the string. If it matches the second character in the *trim_set*, LTRIM removes it, and so on. When the first character in the string does not match the corresponding character in the *trim_set*, LTRIM returns the string and evaluates the next row.

In the example of H. Bender, H does not match either character in the *trim_set* argument, so LTRIM returns the string in the LAST_NAME column and moves to the next row.

### Tips

Use RTRIM and LTRIM with || or CONCAT to remove leading and trailing blanks after you concatenate two strings.

You can also remove multiple sets of characters by nesting LTRIM. For example, if you want to remove leading blanks and the character 'T' from a column of names, you might create an expression similar to the following:

```
LTRIM( LTRIM( NAMES ), 'T' )
```

# MAKE_DATE_TIME

Returns the date and time based on the input values.

### Syntax

```
MAKE_DATE_TIME( year, month, day, hour, minute, second )
```

| Argument | Required/ Optional | Description |
|---|---|---|
| *year* | Required | Numeric datatype. Positive integer. |
| month | Required | Numeric datatype. Positive integer between 1 and 12 (January=1 and December=12). |
| day | Required | Numeric datatype. Positive integer between 1 and 31 (except for the months that have less than 31 days: February, April, June, September, and November). |
| hour | Optional | Numeric datatype. Positive integer between 0 and 24 (where 0=12AM, 12=12PM, and 24 =12AM). |
| minute | Optional | Numeric datatype. Positive integer between 0 and 59. |
| second | Optional | Numeric datatype. Positive integer between 0 and 59. |

### Return Value

Date as MM/DD/YYYY HH24:MI:SS.

### Example

The following expression creates a date and time from the source columns:

```
MAKE_DATE_TIME( SALE_YEAR, SALE_MONTH, SALE_DAY, SALE_HOUR, SALE_MIN, SALE_SEC )
```

| SALE_YR | SALE_MTH | SALE_DAY | SALE_HR | SALE_MIN | SALE_SEC | RETURN VALUE |
|---|---|---|---|---|---|---|
| 2002 | 10 | 27 | 8 | 36 | 22 | 10/27/2002 08:36:22 |
| 2000 | 6 | 15 | 15 | 17 | | 06/15/200 15:17:00 |
| 2003 | 1 | 3 | | 22 | 45 | 01/03/2003 ??:22:45 |
| 04 | 3 | 30 | 12 | 5 | 10 | 03/30/2004 12:05:10 |
| 99 | 12 | 12 | 5 | | 16 | 12/12/1999?? 05:??:16 |

# MAX (Dates)

Returns the latest date found within a field or group. You can apply a filter to limit the rows in the search. You can also use MAX to return the largest numeric value or the highest string value in a field or group.

You can nest only one other aggregate function within MAX. You cannot nest aggregate functions in advanced mode.

Use only in mapping tasks.

### Syntax

```
MAX( date [, filter_condition] )
```

| Argument | Required/ Optional | Description |
|---|---|---|
| *date* | Required | Date/time data type. Passes the date for which you want to return a maximum date. You can enter any valid transformation expression. |
| *filter_condition* | Optional | Limits the rows in the search. The filter condition must be a numeric value or evaluate to TRUE, FALSE, or NULL. You can enter any valid transformation expression. |

### Return Value

Date.

NULL if all values passed to the function are NULL, or if no rows are selected (for example, the filter condition evaluates to FALSE or NULL for all rows).

### Example

You can return the maximum date for a field or group. The following expression returns the maximum order date for flashlights:

```
MAX( ORDERDATE, ITEM_NAME='Flashlight' )
```

| ITEM_NAME | ORDER_DATE |
|---|---|
| Flashlight | Apr 20 1998 |
| Regulator System | May 15 1998 |
| Flashlight | Sep 21 1998 |
| Diving Hood | Aug 18 1998 |
| Flashlight | NULL |

**RETURN VALUE:** Sep 21 1998

# MAX (Numbers)

Returns the maximum numeric value found within a field or group. You can apply a filter to limit the rows in the search. You can also use MAX to return the latest date or the highest string value in a field or group.

You can nest only one other aggregate function within MAX. You cannot nest aggregate functions in advanced mode.

Use only in mapping tasks.

## Syntax

```
MAX( numeric_value [, filter_condition] )
```

| Argument | Required/ Optional | Description |
|---|---|---|
| *numeric_value* | Required | Numeric data type. Passes the numeric values for which you want to return a maximum numeric value. You can enter any valid transformation expression. |
| *filter_condition* | Optional | Limits the rows in the search. The filter condition must be a numeric value or evaluate to TRUE, FALSE, or NULL. You can enter any valid transformation expression. |

## Return Value

Numeric value.

NULL if all values passed to the function are NULL or if no rows are selected (for example, the filter condition evaluates to FALSE or NULL for all rows).

## Nulls

If a value is NULL, MAX ignores it. However, if all values passed from the field are NULL, MAX returns NULL.

## Group By

MAX groups values based on group by fields you define in the transformation, returning one result for each group.

If there is no group by field, MAX treats all rows as one group, returning one value.

## Example

The first expression returns the maximum price for flashlights:

```
MAX( PRICE, ITEM_NAME='Flashlight' )
```

| ITEM_NAME | PRICE |
|---|---|
| Flashlight | 10.00 |
| Regulator System | 360.00 |
| Flashlight | 55.00 |
| Diving Hood | 79.00 |
| Halogen Flashlight | 162.00 |
| Flashlight | 85.00 |
| Flashlight | NULL |

**RETURN VALUE:** 85.00

# MAX (String)

Returns the highest string value found within a field or group. You can apply a filter to limit the rows in the search. You can also use MAX to return the latest date or the largest numeric value in a field or group.

**Note:** The MAX function uses the same sort order that the Sorter transformation uses. However, the MAX function is case sensitive, and the Sorter transformation may not be case sensitive.

You can nest only one other aggregate function within MAX. You cannot nest aggregate functions in advanced mode.

Use only in mapping tasks.

## Syntax

```
MAX( string [, filter_condition] )
```

| Argument | Required/ Optional | Description |
|---|---|---|
| *string* | Required | String data type. Passes the string values for which you want to return a maximum string value. You can enter any valid transformation expression. |
| *filter_condition* | Optional | Limits the rows in the search. The filter condition must be a numeric value or evaluate to TRUE, FALSE, or NULL. You can enter any valid transformation expression. |

## Return Value

String.

NULL if all values passed to the function are NULL, or if no rows are selected (for example, the filter condition evaluates to FALSE or NULL for all rows).

## Nulls

If a value is NULL, MAX ignores it. However, if all values passed from the field are NULL, MAX returns NULL.

## Group By

MAX groups values based on group by fields you define in the transformation, returning one result for each group.

If there is no group by field, MAX treats all rows as one group, returning one value.

## Example

The following expression returns the maximum item name for manufacturer ID 104:

```
MAX( ITEM_NAME, MANUFACTURER_ID='104' )
```

| MANUFACTURER_ID | ITEM_NAME |
|---|---|
| 101 | First Stage Regulator |
| 102 | Electronic Console |
| 104 | Flashlight |
| 104 | Battery (9 volt) |

| MANUFACTURER_ID | ITEM_NAME |
|---|---|
| 104 | Rope (20 ft) |
| 104 | 60.6 cu ft Tank |
| 107 | 75.4 cu ft Tank |
| 108 | Wristband Thermometer |

**RETURN VALUE:** Rope (20 ft)

# MD5

Calculates the checksum of the input value. The function uses Message-Digest algorithm 5 (MD5). MD5 is a one-way cryptographic hash function with a 128-bit hash value. It calculates a unique value for each input. Use MD5 to verify data integrity.

## Syntax

```
MD5( value )
```

| Argument | Required/ Optional | Description |
|---|---|---|
| *value* | Required | String or Binary datatype. Value for which you want to calculate checksum. The case of the input value affects the return value. For example, MD5(informatica) and MD5(Informatica) return different values. |

## Return Value

Unique 32-character string of hexadecimal digits 0-9 and a-f.

NULL if the input is a null value.

## Example

You want to write changed data to a database. You can use the MD5 function to generate a unique checksum value for a row of data each time you read data from a source. When you run new sessions to read data from the same source, you can compare the previously generated checksum value against new checksum values. You can then write rows with new checksum values to the target. Those rows represent data that is changed in the source.

## Tip

You can use the return value as a hash key.

# MEDIAN

Returns the median of all values in a selected field.

If there is an even number of values in the field, the median is the average of the middle two values when all values are placed ordinally on a number line. If there is an odd number of values in the field, the median is the middle number.

You can nest only one other aggregate function within MEDIAN, and the nested function must return a numeric data type. You cannot nest aggregate functions in advanced mode.

Data Integration reads all rows of data to perform the median calculation. The process of reading rows of data to perform the calculation may affect performance. Optionally, you can apply a filter to limit the rows you read to calculate the median.

Use only in mapping tasks.

## Syntax

```
MEDIAN( numeric_value [, filter_condition ] )
```

| Argument | Required/ Optional | Description |
|---|---|---|
| *numeric_value* | Required | Numeric data type. Passes the values for which you want to calculate a median. You can enter any valid transformation expression. |
| *filter_condition* | Optional | Limits the rows in the search. The filter condition must be a numeric value or evaluate to TRUE, FALSE, or NULL. You can enter any valid transformation expression. |

## Return Value

Numeric value.

NULL if all values passed to the function are NULL, or if no rows are selected. For example, the filter condition evaluates to FALSE or NULL for all rows.

## Nulls

If a value is NULL, MEDIAN ignores the row. However, if all values passed from the field are NULL, MEDIAN returns NULL.

## Group By

MEDIAN groups values based on group by fields you define in the transformation, returning one result for each group.

If there is no group by field, MEDIAN treats all rows as one group, returning one value.

## Example

To calculate the median salary for all departments, you create an Aggregator transformation grouped by departments with a field specifying the following expression:

```
MEDIAN( SALARY )
```

The following expression returns the median value for orders of stabilizing vests:

```
MEDIAN( SALES, ITEM = 'Stabilizing Vest' )
```

| ITEM | SALES |
|---|---|
| Flashlight | 85 |
| Stabilizing Vest | 504 |
| Stabilizing Vest | 36 |
| Safety Knife | 5 |
| Medium Titanium Knife | 150 |
| Tank | NULL |
| Stabilizing Vest | 441 |
| Chisel Point Knife | 60 |
| Stabilizing Vest | NULL |
| Stabilizing Vest | 1044 |
| Wrist Band Thermometer | 110 |

**RETURN VALUE:** 472.5

# METAPHONE

Encodes string values. You can specify the length of the string that you want to encode.

METAPHONE encodes characters of the English language alphabet (A-Z). It encodes both uppercase and lowercase letters in uppercase.

METAPHONE encodes characters according to the following list of rules:

- Skips vowels (A, E, I, O, and U) unless one of them is the first character of the input string. METAPHONE('CAR') returns 'KR' and METAPHONE('AAR') returns 'AR'.
- Uses special encoding guidelines.

The following table lists the METAPHONE encoding guidelines:

| Input | Returns | Condition | Example |
|---|---|---|---|
| B | n/a | When it follows M. | METAPHONE ('Lamb') returns LM. |
| B | In all other cases. | METAPHONE ('Box') returns BKS. | |
| C | X | When followed by IA or H. | METAPHONE ('Facial') returns FXL. |

| Input | Returns | Condition | Example |
|---|---|---|---|
| S | When followed by I, E, or Y. | METAPHONE (‘Fence’) returns FNS. | |
| n/a | When it follows S, and is followed by I, E, or Y. | METAPHONE (‘Scene’) returns SN. | |
| K | In all other cases. | METAPHONE (‘Cool’) returns KL. | |
| D | J | When followed by GE, GY, or GI. | METAPHONE (‘Dodge’) returns TJ. |
| T | In all other cases. | METAPHONE (‘David’) returns TFT. | |
| F | F | In all cases. | METAPHONE (‘FOX’) returns FKS. |
| G | F | When followed by H and the first character in the input string is not B, D, or H. | METAPHONE (‘Tough’) returns TF. |
| n/a | When followed by H and the first character in the input string is B, D, or H. | METAPHONE (‘Hugh’) returns HF. | |
| J | When followed by I, E or Y and does not repeat. | METAPHONE (‘Magic’) returns MJK. | |
| K | In all other cases. | METAPHONE(‘GUN’) returns KN. | |
| H | H | When it does not follow C, G, P, S, or T and is followed by A, E, I, or U. | METAPHONE (‘DHAT’) returns THT. |
| n/a | In all other cases. | METAPHONE (‘Chain’) returns XN. | |
| J | J | In all cases. | METAPHONE (‘Jen’) returns JN. |
| K | n/a | When it follows C. | METAPHONE (‘Ckim’) returns KM. |
| K | In all other cases. | METAPHONE (‘Kim’) returns KM. | |
| L | L | In all cases. | METAPHONE (‘Laura’) returns LR. |
| M | M | In all cases. | METAPHONE (‘Maggi’) returns MK. |
| N | N | In all cases. | METAPHONE (‘Nancy’) returns NNS. |
| P | F | When followed by H. | METAPHONE (‘Phone’) returns FN. |
| P | In all other cases. | METAPHONE (‘Pip’) returns PP. | |

| Input | Returns | Condition | Example |
|---|---|---|---|
| Q | K | In all cases. | METAPHONE (‘Queen’) returns KN. |
| R | R | In all cases. | METAPHONE (‘Ray’) returns R. |
| S | X | When followed by H, IO, IA, or CHW. | METAPHONE (‘Cash’) returns KX. |
| S | In all other cases. | METAPHONE (‘Sing’) returns SNK. | |
| T | X | When followed by IA or IO. | METAPHONE (‘Patio’) returns PX. |
| 0 * | When followed by H. | METAPHONE (‘Thor’) returns 0R. | |
| n/a | When followed by CH. | METAPHONE (‘Glitch’) returns KLTX. | |
| T | In all other cases. | METAPHINE (‘Tim’) returns TM. | |
| V | F | In all cases. | METAPHONE (‘Vin’) returns FN. |
| W | W | When followed by A, E, I, O, or U. | METAPHONE (‘Wang’) returns WNK. |
| n/a | In all other cases. | METAPHONE (‘When’) returns HN. | |
| X | KS | In all cases. | METAPHONE (‘Six’) returns SKS. |
| Y | Y | When followed by A, E, I, O, or U. | METAPHONE (‘Yang’) returns YNK. |
| n/a | In all other cases. | METAPHONE (‘Bobby’) returns BB. | |
| Z | S | In all cases. | METAPHONE (‘Zack’) returns SK. |
| * The integer 0. | | | |

- Skips the initial character and encodes the remaining string if the first two characters of the input string have one of the following values:
  - **KN.** For example, METAPHONE(‘KNOT’) returns ‘NT’.
  - **GN.** For example, METAPHONE(‘GNOB’) returns ‘NB’.
  - **PN.** For example, METAPHONE(‘PNRX’) returns ‘NRKS’.
  - **AE.** For example, METAPHONE(‘AERL’) returns ‘ERL’.
- If a character other than “C” occurs more than once in the input string, encodes the first occurrence only. For example, METAPHONE(‘BBOX’) returns ‘BKS’ and METAPHONE(‘CCOX’) returns ‘KKKS’.

## Syntax

```
METAPHONE( string [,length] )
```

| Argument | Required/ Optional | Description |
|---|---|---|
| string | Required | Must be a character string. Passes the value you want to encode. The first character must be a character in the English language alphabet (A-Z). You can enter any valid expression.<br>Skips any non-alphabetic character in *string*. |
| length | Optional | Must be an integer greater than 0. Specifies the number of characters in *string* that you want to encode. You can enter any valid expression.<br>When *length* is 0 or a value greater than the length of *string*, encodes the entire input string.<br>Default is 0. |

## Return Value

String.

NULL if one of the following conditions is true:

- All values passed to the function are NULL.
- No character in *string* is a letter of the English alphabet.
- *string* is empty.

## Examples

The following expression encodes the first two characters in EMPLOYEE_NAME column to a string:

```
METAPHONE( EMPLOYEE_NAME, 2 )
```

| Employee_Name | Return Value |
|---|---|
| John | JH |
| *@#$ | NULL |
| P$%%oc&&KMNL | PK |

The following expression encodes the first four characters in EMPLOYEE_NAME column to a string:

```
METAPHONE( EMPLOYEE_NAME, 4 )
```

| Employee_Name | Return Value |
|---|---|
| John | JHN |
| 1ABC | ABK |
| *@#$ | NULL |
| P$%%oc&&KMNL | PKKM |

# MIN (Dates)

Returns the earliest date found in a field or group. You can apply a filter to limit the rows in the search. You can also use MIN to return the smallest numeric value or the lowest string value in a field or group.

You can nest only one other aggregate function within MIN, and the nested function must return a date data type. You cannot nest aggregate functions in advanced mode.

Use only in mapping tasks.

## Syntax

```
MIN( date [, filter_condition] )
```

| Argument | Required/ Optional | Description |
|---|---|---|
| *date* | Required | Date/time data type. Passes the values for which you want to return minimum value. You can enter any valid transformation expression. |
| *filter_condition* | Optional | Limits the rows in the search. The filter condition must be a numeric value or evaluate to TRUE, FALSE, or NULL. You can enter any valid transformation expression. |

## Return Value

Date if the *value* argument is a date.

NULL if all values passed to the function are NULL, or if no rows are selected (for example, the filter condition evaluates to FALSE or NULL for all rows).

## Nulls

If a single value is NULL, MIN ignores it. However, if all values passed from the field are NULL, MIN returns NULL.

## Group By

MIN groups values based on group by fields you define in the transformation, returning one result for each group.

If there is no group by field, MIN treats all rows as one group, returning one value.

## Example

The following expression returns the oldest order date for flashlights:

```
MIN( ORDER_DATE, ITEM_NAME='Flashlight' )
```

| ITEM_NAME | ORDER_DATE |
|---|---|
| Flashlight | Apr 20 1998 |
| Regulator System | May 15 1998 |
| Flashlight | Sep 21 1998 |
| Diving Hood | Aug 18 1998 |
| Halogen Flashlight | Feb 1 1998 |

| ITEM_NAME | ORDER_DATE |
|---|---|
| Flashlight | Oct 10 1998 |
| Flashlight | NULL |

**RETURN VALUE:** Feb 1 1998

# MIN (Numbers)

Returns the smallest numeric value found in a field or group. You can apply a filter to limit the rows in the search. You can also use MIN to return the latest date or the lowest string value in a field or group.

You can nest only one other aggregate function within MIN, and the nested function must return a numeric data type. You cannot nest aggregate functions in advanced mode.

Use only in mapping tasks.

## Syntax

```
MIN( numeric_value [, filter_condition] )
```

| Argument | Required/ Optional | Description |
|---|---|---|
| *numeric_value* | Required | Numeric data types. Passes the values for which you want to return minimum value. You can enter any valid transformation expression. |
| *filter_condition* | Optional | Limits the rows in the search. The filter condition must be a numeric value or evaluate to TRUE, FALSE, or NULL. You can enter any valid transformation expression. |

## Return Value

Numeric value.

NULL if all values passed to the function are NULL, or if no rows are selected (for example, the filter condition evaluates to FALSE or NULL for all rows).

## Nulls

If a single value is NULL, MIN ignores it. However, if all values passed from the field are NULL, MIN returns NULL.

## Group By

MIN groups values based on group by fields you define in the transformation, returning one result for each group.

If there is no group by field, MIN treats all rows as one group, returning one value.

### Example

The following expression returns the minimum price for flashlights:

```
MIN ( PRICE, ITEM_NAME='Flashlight' )
```

| ITEM_NAME | PRICE |
|---|---|
| Flashlight | 10.00 |
| Regulator System | 360.00 |
| Flashlight | 55.00 |
| Diving Hood | 79.00 |
| Halogen Flashlight | 162.00 |
| Flashlight | 85.00 |
| Flashlight | NULL |

**RETURN VALUE:** 10.00

# MIN (String)

Returns the lowest string value found in a field or group. You can apply a filter to limit the rows in the search.

**Note:** The MIN function uses the same sort order that the Sorter transformation uses. However, the MIN function is case sensitive, but the Sorter transformation may not be case sensitive.

You can also use MIN to return the latest date or the minimum numeric value in a field or group.

You can nest only one other aggregate function within MIN, and the nested function must return a string data type. You cannot nest aggregate functions in advanced mode.

Use only in mapping tasks.

### Syntax

```
MIN( string [, filter_condition] )
```

| Argument | Required/ Optional | Description |
|---|---|---|
| *string* | Required | String data type. Passes the values for which you want to return minimum value. You can enter any valid transformation expression. |
| *filter_condition* | Optional | Limits the rows in the search. The filter condition must be a numeric value or evaluate to TRUE, FALSE, or NULL. You can enter any valid transformation expression. |

### Return Value

String value.

NULL if all values passed to the function are NULL, or if no rows are selected (for example, the filter condition evaluates to FALSE or NULL for all rows).

## Nulls

If a single value is NULL, MIN ignores it. However, if all values passed from the field are NULL, MIN returns NULL.

## Group By

MIN groups values based on group by fields you define in the transformation, returning one result for each group.

If there is no group by field, MIN treats all rows as one group, returning one value.

## Example

The following expression returns the minimum item name for manufacturer ID 104:

```
MIN ( ITEM_NAME, MANUFACTURER_ID='104' )
```

| MANUFACTURER_ID | ITEM_NAME |
|---|---|
| 101 | First Stage Regulator |
| 102 | Electronic Console |
| 104 | Flashlight |
| 104 | Battery (9 volt) |
| 104 | Rope (20 ft) |
| 104 | 60.6 cu ft Tank |
| 107 | 75.4 cu ft Tank |
| 108 | Wristband Thermometer |

**RETURN VALUE:** 60.6 cu ft Tank

# MOD

Returns the remainder of a division calculation. For example, `MOD(8,5)` returns 3.

## Syntax

```
MOD( numeric_value, divisor )
```

| Argument | Required/ Optional | Description |
|---|---|---|
| *numeric_value* | Required | Numeric datatype. The values you want to divide. You can enter any valid expression. |
| *divisor* | Required | The numeric value you want to divide by. The divisor cannot be 0. |

## Return Value

Numeric value of the datatype you pass to the function. The remainder of the numeric value divided by the divisor.

NULL if a value passed to the function is NULL.

## Example

The following expression returns the modulus of the values in the PRICE column divided by the values in the QTY column:

```
MOD( PRICE, QTY )
```

| PRICE | QTY | RETURN VALUE |
|---|---|---|
| 10.00 | 2 | 0 |
| 12.00 | 5 | 2 |
| 9.00 | 2 | 1 |
| 15.00 | 3 | 0 |
| NULL | 3 | NULL |
| 20.00 | NULL | NULL |
| 25.00 | 0 | None. Data Integration writes the row into the error rows file. |

The last row (25, 0) produced an error because you cannot divide by 0. To avoid dividing by 0, you can create an expression similar to the following, which returns the modulus of Price divided by Quantity only if the quantity is not 0. If the quantity is 0, the function returns NULL:

```
MOD( PRICE, IIF( QTY = 0, NULL, QTY ))
```

| PRICE | QTY | RETURN VALUE |
|---|---|---|
| 10.00 | 2 | 0 |
| 12.00 | 5 | 2 |
| 9.00 | 2 | 1 |

| PRICE | QTY | RETURN VALUE |
|---|---|---|
| 15.00 | 3 | 0 |
| NULL | 3 | NULL |
| 20.00 | NULL | NULL |
| 25.00 | 0 | NULL |

The last row (25, 0) produced a NULL rather than an error because the IIF function replaces NULL with the 0 in the QTY column.

# MOVINGAVG

Returns the average (row-by-row) of a specified set of rows. Optionally, you can apply a condition to filter rows before calculating the moving average.

## Syntax

```
MOVINGAVG( numeric_value, rowset [, filter_condition] )
```

| Argument | Required/ Optional | Description |
|---|---|---|
| *numeric_value* | Required | Numeric data type. The values for which you want to calculate a moving average. You can enter any valid transformation expression. |
| *rowset* | Required | Must be a positive integer literal greater than 0. Defines the row set for which you want to calculate the moving average. For example, if you want to calculate a moving average for a column of data, five rows at a time, you might write an expression such as: `MOVINGAVG(SALES, 5)`. |
| *filter_condition* | Optional | Limits the rows in the search. The filter condition must be a numeric value or evaluate to TRUE, FALSE, or NULL. You can enter any valid transformation expression. |

## Return Value

Numeric value.

NULL if all values passed to the function are NULL or if no rows are selected (for example, the filter condition evaluates to FALSE or NULL for all rows).

**Note:** If the return value is Decimal with precision greater than 15, you can enable high precision to ensure decimal precision up to 28 digits.

## Nulls

MOVINGAVG ignores null values when calculating the moving average. However, if all values are NULL, the function returns NULL.

### Example

The following expression returns the average order for a Stabilizing Vest, based on the first five rows in the SALES column, and thereafter, returns the average for the last five rows read:

```
MOVINGAVG( SALES, 5 )
```

| ROW_NO | SALES | RETURN VALUE |
|---|---|---|
| 1 | 600 | NULL |
| 2 | 504 | NULL |
| 3 | 36 | NULL |
| 4 | 100 | NULL |
| 5 | 550 | 358 |
| 6 | 39 | 245.8 |
| 7 | 490 | 243 |

The function returns the average for a set of five rows: 358 based on rows 1 through 5, 245.8 based on rows 2 through 6, and 243 based on rows 3 through 7.

# MOVINGSUM

Returns the sum (row-by-row) of a specified set of rows. Optionally, you can apply a condition to filter rows before calculating the moving sum.

### Syntax

```
MOVINGSUM( numeric_value, rowset [, filter_condition] )
```

| Argument | Required/ Optional | Description |
|---|---|---|
| *numeric_value* | Required | Numeric data type. The values for which you want to calculate a moving sum. You can enter any valid transformation expression. |
| *rowset* | Required | Must be a positive integer literal greater than 0. Defines the rowset for which you want to calculate the moving sum. For example, if you want to calculate a moving sum for a column of data, five rows at a time, you might write an expression such as: `MOVINGSUM( SALES, 5 )` |
| *filter_condition* | Optional | Limits the rows in the search. The filter condition must be a numeric value or evaluate to TRUE, FALSE, or NULL. You can enter any valid transformation expression. |

### Return Value

Numeric value.

NULL if all values passed to the function are NULL, or if the function does not select any rows (for example, the filter condition evaluates to FALSE or NULL for all rows).

**Note:** If the return value is Decimal with precision greater than 15, you can enable high precision to ensure decimal precision up to 28 digits.

### Nulls

MOVINGSUM ignores null values when calculating the moving sum. However, if all values are NULL, the function returns NULL.

### Example

The following expression returns the sum of orders for a Stabilizing Vest, based on the first five rows in the SALES column, and thereafter, returns the average for the last five rows read:

```
MOVINGSUM( SALES, 5 )
```

| ROW_NO | SALES | RETURN VALUE |
|---|---|---|
| 1 | 600 | NULL |
| 2 | 504 | NULL |
| 3 | 36 | NULL |
| 4 | 100 | NULL |
| 5 | 550 | 1790 |
| 6 | 39 | 1229 |
| 7 | 490 | 1215 |

The function returns the sum for a set of five rows: 1790 based on rows 1 through 5, 1229 based on rows 2 through 6, and 1215 based on rows 3 through 7.

## NPER

Returns the number of periods for an investment based on a constant interest rate and periodic, constant payments.

### Syntax

```
NPER( rate, present value, payment [, future value, type] )
```

| Argument | Required/ Optional | Description |
|---|---|---|
| rate | Required | Numeric. Interest rate earned in each period. Expressed as a decimal number. Divide the rate by 100 to express it as a decimal number. Must be greater than or equal to 0. |
| present value | Required | Numeric. Lump-sum amount a series of future payments is worth. |

| Argument | Required/ Optional | Description |
|---|---|---|
| payment | Required | Numeric. Payment amount due per period. Must be a negative number. |
| future value | Optional | Numeric. Cash balance you want to attain after the last payment is made. If you omit this value, NPER uses 0. |
| type | Optional | Boolean. Timing of the payment. Enter 1 if payment is at the beginning of period. Enter 0 if payment is at the end of period. Default is 0. If you enter a value other than 0 or 1, Data Integration treats the value as 1. |

### Return Value

Numeric.

### Example

The present value of an investment is $500. Each payment is $2000 and the future value of the investment is $20,000. The following expression returns 9 as the number of periods for which you need to make the payments:

```
NPER( 0.015, -500, -2000, 20000, TRUE )
```

### Notes

To calculate interest rate earned in each period, divide the annual rate by the number of payments made in an year. For example, if you make monthly payments at an annual interest rate of 15 percent, the value of the Rate argument is 15% divided by 12. If you make annual payments, the value of the Rate argument is 15%.

The payment value and present value are negative because these are amounts that you pay.

# PERCENTILE

Calculates the value that falls at a given percentile in a group of numbers.

Data Integration reads all rows of data to perform the percentile calculation. The process of reading rows to perform the calculation may affect performance. Optionally, you can apply a filter to limit the rows you read to calculate the percentile.

You can nest only one other aggregate function within PERCENTILE, and the nested function must return a numeric data type. You cannot nest aggregate functions in advanced mode.

Use only in mapping tasks.

## Syntax

```
PERCENTILE( numeric_value, percentile [, filter_condition ] )
```

| Argument | Required/ Optional | Description |
|---|---|---|
| *numeric_value* | Required | Numeric data type. Passes the values for which you want to calculate a percentile. You can enter any valid transformation expression. |
| *percentile* | Required | Integer between 0 and 100, inclusive. Passes the percentile you want to calculate. You can enter any valid transformation expression. If you pass a number outside the 0 to 100 range, Data Integration displays an error and does not write the row. |
| *filter_condition* | Optional | Limits the rows in the search. The filter condition must be a numeric value or evaluate to TRUE, FALSE, or NULL. You can enter any valid transformation expression. |

## Return Value

Numeric value.

NULL if all values passed to the function are NULL, or if no rows are selected (for example, the filter condition evaluates to FALSE or NULL for all rows).

## Nulls

If a value is NULL, PERCENTILE ignores the row. However, if all values in a group are NULL, PERCENTILE returns NULL.

## Group By

PERCENTILE groups values based on group by fields you define in the transformation, returning one result for each group.

If there is no group by field, PERCENTILE treats all rows as one group, returning one value.

## Example

Data Integration calculates a percentile using the following logic:

$$i = \frac{(x + 1) \times \text{percentile}}{100}$$

Use the following guidelines for this equation:

- *x* is the number of elements in the group of values for which you are calculating a percentile.
- If *i* < 1, PERCENTILE returns the value of the first element in the list.
- If *i* is an integer value, PERCENTILE returns the value of the *i*th element in the list.
- Otherwise PERCENTILE returns the value of *n*:

$$n = [\lfloor i \rfloor \text{th?element} \times (\lceil i \rceil - i)] + [\lceil i \rceil \text{th?element} \times (i - \lfloor i \rfloor)]$$

The following expression returns the salary that falls at the 75th percentile of salaries greater than $50,000:

```
PERCENTILE( SALARY, 75, SALARY > 50000 )
```

```
SALARY
125000.0
27900.0
100000.0
NULL
55000.0
9000.0
85000.0
86000.0
48000.0
99000.0
RETURN VALUE:  106250.0
```

# PMT

Returns the payment for a loan based on constant payments and a constant interest rate.

## Syntax

```
PMT( rate, terms, present value, future value, type] )
```

| Argument | Required/ Optional | Description |
|---|---|---|
| rate | Required | Numeric. Interest rate of the loan for each period. Expressed as a decimal number. Divide the rate by 100 to express it as a decimal number. Must be greater than or equal to 0. |
| terms | Required | Numeric. Number of periods or payments. Must be greater than 0. |
| present value | Required | Numeric. Principal for the loan. |
| future value | Optional | Numeric. Cash balance you want to attain after the last payment. If you omit this value, PMT uses 0. |
| type | Optional | Boolean. Timing of the payment. Enter 1 if payment is at the beginning of period. Enter 0 if payment is at the end of period. Default is 0. If you enter a value other than 0 or 1, Data Integration treats the value as 1. |

### Return Value

Numeric.

### Example

The following expression returns -2111.64 as the monthly payment amount of a loan:

```
PMT( 0.01, 10, 20000 )
```

### Notes

To calculate interest rate earned in each period, divide the annual rate by the number of payments made in a year. For example, if you make monthly payments at an annual interest rate of 15%, the rate is 15%/12. If you make annual payments, the rate is 15%.

The payment value is negative because these are amounts that you pay.

# POWER

Returns a value raised to the exponent you pass to the function.

### Syntax

```
POWER( base, exponent )
```

| Argument | Required/ Optional | Description |
|---|---|---|
| *base* | Required | Numeric value. This argument is the base value. You can enter any valid expression. If the base value is negative, the exponent must be an integer. |
| *exponent* | Required | Numeric value. This argument is the exponent value. You can enter any valid expression. If the base value is negative, the exponent must be an integer. In this case, the function rounds any decimal values to the nearest integer before returning a value. |

### Return Value

Double value.

NULL if you pass a null value to the function.

### Example

The following expression returns the values in the Numbers column raised to the values in the Exponent column:

```
POWER( NUMBERS, EXPONENT )
```

| NUMBERS | EXPONENT | RETURN VALUE |
|---|---|---|
| 10.0 | 2.0 | 100 |
| 3.5 | 6.0 | 1838.265625 |
| 3.5 | 5.5 | 982.594307804838 |
| NULL | 2.0 | NULL |

| NUMBERS | EXPONENT | RETURN VALUE |
|---|---|---|
| 10.0 | NULL | NULL |
| -3.0 | -6.0 | 0.00137174211248285 |
| 3.0 | -6.0 | 0.00137174211248285 |
| -3.0 | 6.0 | 729.0 |
| -3.0 | 5.5 | 729.0 |

The value -3.0 raised to 6 returns the same results as -3.0 raised to 5.5. If the base is negative, the exponent must be an integer. Otherwise, Data Integration rounds the exponent to the nearest integer value.

# PV

Returns the present value of an investment.

## Syntax

```
PV( rate, terms, payment [, future value, type] )
```

| Argument | Required/ Optional | Description |
|---|---|---|
| rate | Required | Numeric. Interest rate earned in each period. Expressed as a decimal number. Divide the rate by 100 to express it as a decimal number. Must be greater than or equal to 0. |
| terms | Required | Numeric. Number of periods or payments. Must be greater than 0. |
| payment | Required | Numeric. Payment amount due per period. Must be a negative number. |
| future value | Optional | Numeric. Cash balance after the last payment. If you omit this value, PV uses 0. |
| type | Optional | Boolean. Timing of the payment. Enter 1 if payment is at the beginning of period. Enter 0 if payment is at the end of period. Default is 0. If you enter a value other than 0 or 1, Data Integration treats the value as 1. |

## Return Value

Numeric.

## Example

The following expression returns 12,524.43 as the amount you must deposit in the account today to have a future value of $20,000 in one year if you also deposit $500 at the beginning of each period:

```
PV( 0.0075, 12, -500, 20000, TRUE )
```

# RAND

Returns a random number between 0 and 1. This is useful for probability calculations.

## Syntax

```
RAND( seed )
```

| Argument | Required/ Optional | Description |
|---|---|---|
| seed | Optional | Numeric. Starting value for Data Integration to generate the random number. Value must be a constant. If you do not enter a seed, Data Integration uses the current system time to derive the numbers of seconds since January 1, 1971. It uses this value as the seed. |

## Return Value

Numeric.

For the same seed, Data Integration generates the same sequence of numbers.

## Example

The following expression may return a value of 0.417022004702574:

```
RAND (1)
```

# RATE

Returns the interest rate earned for each period by a security.

## Syntax

```
RATE( terms, payment, present value[, future value, type] )
```

| Argument | Required/ Optional | Description |
|---|---|---|
| terms | Required | Numeric. Number of periods or payments. Must be greater than 0. |
| payment | Required | Numeric. Payment amount due for each period. Must be a negative number. |
| present value | Required | Numeric. Lump-sum amount that a series of future payments is worth now. |
| future value | Optional | Numeric. Cash balance you want to attain after the last payment. For example, the future value of a loan is 0. If you omit this argument, RATE uses 0. |
| type | Optional | Boolean. Timing of the payment. Enter 1 if payment is at the beginning of period. Enter 0 if payment is at the end of period. Default is 0. If you enter a value other than 0 or 1, Data Integration treats the value as 1. |

## Return Value

Numeric.

### Example

The following expression returns 0.0077 as the monthly interest rate of a loan:

```
RATE( 48, -500, 20000 )
```

To calculate the annual interest rate of the loan, multiply 0.0077 by 12. The annual interest rate is 0.0924 or 9.24%.

# REG_EXTRACT

Extracts subpatterns of a regular expression within an input value. For example, from a regular expression pattern for a full name, you can also extract the first name or last name.

**Note:** Use the REG_REPLACE function to replace a character pattern in a string with another character pattern.

### Syntax

```
REG_EXTRACT (subject, pattern, subPatternNum)
```

| Argument | Required/ Optional | Description |
|---|---|---|
| subject | Required | String datatype. Passes the value you want to compare against the regular expression pattern. |
| pattern | Required | String datatype. Regular expression pattern that you want to match. You must use perl compatible regular expression syntax. Enclose the pattern in single quotation marks. |
| subPatternNum | Optional | Integer value. Subpattern number of the regular expression you want to match. Use the following guidelines to determine the subpattern number:<br>- no value or 1. Extracts the first regular expression subpattern.<br>- 2. Extracts the second regular expression subpattern.<br>- n. Extracts the nth regular expression subpattern.<br>Default is 1. |

### Using perl Compatible Regular Expression Syntax

You must use perl compatible regular expression syntax with REG_EXTRACT, REG_MATCH and REG_REPLACE functions.

The following table provides perl compatible regular expression syntax guidelines:

| Syntax | Description |
|---|---|
| . (period) | Matches any one character. |
| [a-z] | Matches one instance of a character in lower case. For example, [a-z] matches ab. Use [A-Z] to match characters in upper case. |
| \d | Matches one instance of any digit from 0-9. |
| \s | Matches a whitespace character. |

| Syntax | Description |
|---|---|
| \w | Matches one alphanumeric character, including underscore (_) |
| () | Groups an expression. For example, the parentheses in (\d-\d-\d\d) groups the expression \d\d-\d\d, which finds any two numbers followed by a hyphen and any two numbers, as in 12-34. |
| {} | Matches the number of characters. For example, \d{3} matches any three numbers, such as 650 or 510. Or, [a-z]{2} matches any two letters, such as CA or NY. |
| ? | Matches the preceding character or group of characters zero or one time. For example, \d{3}(-{d{4})? matches any three numbers, which can be followed by a hyphen and any four numbers. |
| * (asterisk) | Matches zero or more instances of the values that follow the asterisk. For example, *0 is any value that precedes a 0. |
| + | Matches one or more instances of the values that follow the plus sign. For example, \w+ is any value that follows an alphanumeric character. |

For example, the following regular expression finds 5-digit U.S.A. zip codes, such as 93930, and 9-digit zip codes, such as 93930-5407:

```
\d{5}(-\d{4})?
```

\d{5} refers to any five numbers, such as 93930. The parentheses surrounding -\d{4} group this segment of the expression. The hyphen represents the hyphen of a 9-digit zip code, as in 93930-5407. \d{4} refers to any four numbers, such as 5407. The question mark states that the hyphen and last four digits are optional or can appear one time.

## Converting COBOL Syntax to perl Compatible Regular Expression Syntax

If you are familiar with COBOL syntax, you can use the following information to write perl compatible regular expressions.

The following table shows examples of COBOL syntax and their perl equivalents:

| COBOL Syntax | perl Syntax | Description |
|---|---|---|
| 9 | \d | Matches one instance of any digit from 0-9. |
| 9999 | \d\d\d\d<br>or<br>\d{4} | Matches any four digits from 0-9, as in 1234 or 5936. |
| x | [a-z] | Matches one instance of a letter. |
| 9xx9 | \d[a-z][a-z]\d | Matches any number followed by two letters and another number, as in 1ab2. |

## Converting SQL Syntax to perl Compatible Regular Expression Syntax

If you are familiar with SQL syntax, you can use the following information to write perl compatible regular expressions.

The following table shows examples of SQL syntax and their perl equivalents:

| SQL Syntax | perl Syntax | Description |
|---|---|---|
| % | . * | Matches any string. |
| A% | A.* | Matches the letter “A” followed by any string, as in Area. |
| _ | . (a period) | Matches any one character. |
| A_ | A. | Matches “A” followed by any one character, such as AZ. |

### Return Value

Returns the value of the *n*th subpattern that is part of the input value. The *nth* subpattern is based on the value you specify for subPatternNum.

NULL if the input is a null value or if the pattern is null.

### Example

You might use REG_EXTRACT in an expression to extract middle names from a regular expression that matches first name, middle name, and last name. For example, the following expression returns the middle name of a regular expression:

```
REG_EXTRACT (Employee_Name, '(\w+)\s+(\w+)\s+(\w+)', 2)
```

| Employee_Name | Return Value |
|---|---|
| Stephen Graham Smith | Graham |
| Juan Carlos Fernando | Carlos |

# REG_MATCH

Returns whether a value matches a regular expression pattern. This lets you validate data patterns, such as IDs, telephone numbers, postal codes, and state names.

**Note:** Use the REG_REPLACE function to replace a character pattern in a string with a new character pattern.

### Syntax

```
REG_MATCH( subject,pattern )
```

| Argument | Required/ Optional | Description |
|---|---|---|
| subject | Required | String datatype. Passes the value you want to match against the regular expression pattern. |
| pattern | Required | String datatype. Regular expression pattern that you want to match. You must use perl compatible regular expression syntax. Enclose the pattern in single quotation marks. For more information, see “REG_EXTRACT” on page 159. |

## Return Value

1 if the data matches the pattern.

0 if the data does not match the pattern.

NULL if the input is a null value or if the pattern is NULL.

## Example

You might use REG_MATCH in an expression to validate telephone numbers. For example, the following expression matches a 10-digit telephone number against the pattern and returns a numerical value based on the match:

```
REG_MATCH (Phone_Number, '(\d\d\d-\d\d\d-\d\d\d\d)' )
```

| Phone_Number | Return Value |
|---|---|
| 408-555-1212 | 1 |
| | NULL |
| 510-555-1212 | 1 |
| 92 555 51212 | 0 |
| 650-555-1212 | 1 |
| 415-555-1212 | 1 |
| 831 555 12123 | 0 |

## Tip

You can also use REG_MATCH for the following tasks:

- To verify that a value matches a pattern. This use is similar to the SQL LIKE function.
- To verify that values are characters. This use is similar to the SQL IS_CHAR function.

To verify that a value matches a pattern, use a period (.) and an asterisk (*) with the REG_MATCH function in an expression. A period matches any one character. An asterisk matches 0 or more instances of values that follow it.

For example, use the following expression to find account numbers that begin with 1835:

```
REG_MATCH(ACCOUNT_NUMBER, '1835.*')
```

To verify that values are characters, use a REG_MATCH function with the regular expression [a-zA-Z]+. a-z matches all lowercase characters. A-Z matches all uppercase characters. The plus sign (+) indicates that there should be at least one character.

For example, use the following expression to verify that a list of last names contain only characters:

```
REG_MATCH(LAST_NAME, '[a-zA-Z]+')
```

# REG_REPLACE

Replaces characters in a string with a another character pattern. By default, REG_REPLACE searches the input string for the character pattern you specify and replaces all occurrences with the replacement pattern. You can also indicate the number of occurrences of the pattern you want to replace in the string.

## Syntax

```
REG_REPLACE( subject, pattern, replace, numReplacements )
```

| Argument | Required/ Optional | Description |
|---|---|---|
| *subject* | Required | String datatype. Passes the string you want to search. |
| *pattern* | Required | String datatype. Passes the character string to be replaced. You must use perl compatible regular expression syntax. Enclose the pattern in single quotation marks. For more information, see "REG_EXTRACT" on page 159. |
| replace | Required | String datatype. Passes the new character string. |
| numReplacements | Optional | Numeric datatype. Specifies the number of occurrences you want to replace. If you omit this option, REG_REPLACE will replace all occurrences of the character string. |

## Return Value

String.

## Example

The following expression removes additional spaces from the Employee name data for each row of the Employee_name column:

```
REG_REPLACE( Employee_Name, '\s+', ' ')
```

| Employee_Name | RETURN VALUE |
|---|---|
| Adam Smith | Adam Smith |
| Greg  Sanders | Greg Sanders |
| Sarah  Fe | Sarah Fe |
| Sam  Cooper | Sam Cooper |

# REPLACECHR

Replaces characters in a string with a single character or no character. REPLACECHR searches the input string for the characters you specify and replaces all occurrences of all characters with the new character you specify.

## Syntax

```
REPLACECHR( CaseFlag, InputString, OldCharSet, NewChar )
```

| Argument | Required/ Optional | Description |
|---|---|---|
| *CaseFlag* | Required | Must be an integer. Determines whether the arguments in this function are case sensitive. You can enter any valid expression.<br>When *CaseFlag* is a number other than 0, the function is case sensitive.<br>When *CaseFlag* is a null value or 0, the function is not case sensitive. |
| *InputString* | Required | Must be a character string. Passes the string you want to search. You can enter any valid expression. If you pass a numeric value, the function converts it to a character string.<br>If *InputString* is NULL, REPLACECHR returns NULL. |
| OldCharSet | Required | Must be a character string. The characters you want to replace. You can enter one or more characters. You can enter any valid expression. You can also enter a text literal enclosed within single quotation marks, for example, 'abc'.<br>If you pass a numeric value, the function converts it to a character string.<br>If *OldCharSet* is NULL or empty, REPLACECHR returns *InputString*. |
| NewChar | Required | Must be a character string. You can enter one character, an empty string, or NULL. You can enter any valid expression.<br>If *NewChar* is NULL or empty, REPLACECHR removes all occurrences of all characters in *OldCharSet* in *InputString*.<br>If *NewChar* contains more than one character, REPLACECHR uses the first character to replace *OldCharSet*. |

## Return Value

String.

Empty string if REPLACECHR removes all characters in *InputString*.

NULL if *InputString* is NULL.

*InputString* if *OldCharSet* is NULL or empty.

## Example

The following expression removes the double quotation marks from web log data for each row in the WEBLOG column:

```
REPLACECHR( 0, WEBLOG, '"', NULL )
```

| WEBLOG | RETURN VALUE |
|---|---|
| "GET /news/index.html HTTP/1.1" | GET /news/index.html HTTP/1.1 |
| "GET /companyinfo/index.html HTTP/1.1" | GET /companyinfo/index.html HTTP/1.1 |
| GET /companyinfo/index.html HTTP/1.1 | GET /companyinfo/index.html HTTP/1.1 |

| WEBLOG | RETURN VALUE |
|---|---|
| NULL | NULL |

The following expression removes multiple characters for each row in the WEBLOG column:

```
REPLACECHR ( 1, WEBLOG, ']["', NULL )
```

| WEBLOG | RETURN VALUE |
|---|---|
| [29/Oct/2001:14:13:50 -0700] | 29/Oct/2001:14:13:50 -0700 |
| [31/Oct/2000:19:45:46 -0700] "GET /news/index.html HTTP/1.1" | 31/Oct/2000:19:45:46 -0700 GET /news/index.html HTTP/1.1 |
| [01/Nov/2000:10:51:31 -0700] "GET /news/index.html HTTP/1.1" | 01/Nov/2000:10:51:31 -0700 GET /news/index.html HTTP/1.1 |
| NULL | NULL |

The following expression changes part of the value of the customer code for each row in the CUSTOMER_CODE column:

```
REPLACECHR ( 1, CUSTOMER_CODE, 'A', 'M' )
```

| CUSTOMER_CODE | RETURN VALUE |
|---|---|
| ABA | MBM |
| abA | abM |
| BBC | BBC |
| ACC | MCC |
| NULL | NULL |

The following expression changes part of the value of the customer code for each row in the CUSTOMER_CODE column:

```
REPLACECHR ( 0, CUSTOMER_CODE, 'A', 'M' )
```

| CUSTOMER_CODE | RETURN VALUE |
|---|---|
| ABA | MBM |
| abA | MbM |
| BBC | BBC |
| ACC | MCC |

The following expression changes part of the value of the customer code for each row in the CUSTOMER_CODE column:

```
REPLACECHR ( 1, CUSTOMER_CODE, 'A', NULL )
```

| CUSTOMER_CODE | RETURN VALUE |
|---|---|
| ABA | B |

| CUSTOMER_CODE | RETURN VALUE |
|---|---|
| BBC | BBC |
| ACC | CC |
| AAA | *[empty string]* |
| aaa | aaa |
| NULL | NULL |

The following expression removes multiple numbers for each row in the INPUT column:

```
REPLACECHR ( 1, INPUT, '14', NULL )
```

| INPUT | RETURN VALUE |
|---|---|
| 12345 | 235 |
| 4141 | NULL |
| 111115 | 5 |
| NULL | NULL |

When you want to use a single quotation mark (') in either *OldCharSet* or *NewChar*, you must use the CHR function. The single quotation mark is the only character that cannot be used inside a string literal.

The following expression removes multiple characters, including the single quotation mark, for each row in the INPUT column:

```
REPLACECHR (1, INPUT, CHR(39), NULL )
```

| INPUT | RETURN VALUE |
|---|---|
| 'Tom Smith' 'Laura Jones' | Tom Smith Laura Jones |
| Tom's | Toms |
| NULL | NULL |

# REPLACESTR

Replaces characters in a string with a single character, multiple characters, or no character. REPLACESTR searches the input string for all strings you specify and replaces them with the new string you specify.

## Syntax

```
REPLACESTR ( CaseFlag, InputString, OldString1, [OldString2, ... OldStringN,] NewString )
```

| Argument | Required/ Optional | Description |
|---|---|---|
| *CaseFlag* | Required | Must be an integer. Determines whether the arguments in this function are case sensitive. You can enter any valid expression.<br>When *CaseFlag* is a number other than 0, the function is case sensitive.<br>When *CaseFlag* is a null value or 0, the function is not case sensitive. |
| *InputString* | Required | Must be a character string. Passes the strings you want to search. You can enter any valid expression. If you pass a numeric value, the function converts it to a character string.<br>If *InputString* is NULL, REPLACESTR returns NULL. |
| *OldString* | Required | Must be a character string. The string you want to replace. You must enter at least one *OldString* argument. You can enter one or more characters per *OldString* argument. You can enter any valid expression. You can also enter a text literal enclosed within single quotation marks, for example, 'abc'.<br>If you pass a numeric value, the function converts it to a character string.<br>When REPLACESTR contains multiple *OldString* arguments, and one or more *OldString* arguments is NULL or empty, REPLACESTR ignores the *OldString* argument. When all *OldString* arguments are NULL or empty, REPLACESTR returns *InputString*.<br>The function replaces the characters in the *OldString* arguments in the order they appear in the function. For example, if you enter multiple *OldString* arguments, the first *OldString* argument has precedence over the second *OldString* argument, and the second *OldString* argument has precedence over the third *OldString* argument. When REPLACESTR replaces a string, it places the cursor after the replaced characters in *InputString* before searching for the next match. For more information, see the examples. |
| *NewString* | Required | Must be a character string. You can enter one character, multiple characters, an empty string, or NULL. You can enter any valid expression.<br>If *NewString* is NULL or empty, REPLACESTR removes all occurrences of *OldString* in *InputString*. |

## Return Value

String.

Empty string if REPLACESTR removes all characters in *InputString*.

NULL if *InputString* is NULL.

*InputString* if all *OldString* arguments are NULL or empty.

## Example

The following expression removes double quotation marks and two different text strings from web log data for each row in the WEBLOG column:

```
REPLACESTR( 1, WEBLOG, '"', 'GET ', ' HTTP/1.1', NULL )
```

| WEBLOG | RETURN VALUE |
|---|---|
| "GET /news/index.html HTTP/1.1" | /news/index.html |
| "GET /companyinfo/index.html HTTP/1.1" | /companyinfo/index.html |
| GET /companyinfo/index.html | /companyinfo/index.html |
| GET | *[empty string]* |
| NULL | NULL |

The following expression changes the title for certain values for each row in the TITLE column:

```
REPLACESTR ( 1, TITLE, 'rs.', 'iss', 's.' )
```

| TITLE | RETURN VALUE |
|---|---|
| Mrs. | Ms. |
| Miss | Ms. |
| Mr. | Mr. |
| MRS. | MRS. |

The following expression changes the title for certain values for each row in the TITLE column:

```
REPLACESTR ( 0, TITLE, 'rs.', 'iss', 's.' )
```

| TITLE | RETURN VALUE |
|---|---|
| Mrs. | Ms. |
| MRS. | Ms. |

The following expression shows how the REPLACESTR function replaces multiple *OldString* arguments for each row in the INPUT column:

```
REPLACESTR ( 1, INPUT, 'ab', 'bc', '*' )
```

| INPUT | RETURN VALUE |
|---|---|
| abc | *c |
| abbc | ** |
| abbbbc | *bb* |
| bc | * |

The following expression shows how the REPLACESTR function replaces multiple *OldString* arguments for each row in the INPUT column:

```
REPLACESTR ( 1, INPUT, 'ab', 'bc', 'b' )
```

| INPUT | RETURN VALUE |
|---|---|
| ab | b |
| bc | b |
| abc | bc |
| abbc | bb |
| abbcc | bbc |

When you want to use a single quotation mark (') in either *OldString* or *NewString*, you must use the CHR function. Use both the CHR and CONCAT functions to concatenate a single quotation mark onto a string. The single quotation mark is the only character that cannot be used inside a string literal. Consider the following example:

```
CONCAT( 'Joan', CONCAT( CHR(39), 's car' ))
```

The return value is:

```
Joan's car
```

The following expression changes a string that includes the single quotation mark, for each row in the INPUT column:

```
REPLACESTR ( 1, INPUT, CONCAT('it', CONCAT(CHR(39), 's' )), 'its' )
```

| INPUT | RETURN VALUE |
|---|---|
| it's | its |
| mit's | mits |
| mits | mits |
| mits' | mits' |

# REVERSE

Reverses the input string.

## Syntax

```
REVERSE( string )
```

| Argument | Required/ Optional | Description |
|---|---|---|
| string | Required | Any character value. Value you want to reverse. |

### Return Value

String. Reverse of the input value.

### Example

The following expression reverses the numbers of the customer code:

```
REVERSE( CUSTOMER_CODE )
```

| CUSTOMER_CODE | RETURN VALUE |
|---|---|
| 0001 | 1000 |
| 0002 | 2000 |
| 0003 | 3000 |
| 0004 | 4000 |

# ROUND (Dates)

Rounds one part of a date. You can also use ROUND to round numbers.

This functions can round the following parts of a date:

- **Year.** Rounds the year portion of a date based on the month. If the month is between January and June, the function returns January 1 of the input year, and sets the time to 00:00:00. If the month is between July and December, the function returns January 1 of the next year with the time set to 00:00:00. For example, the expression `ROUND(06/30/1998 2:30:55, 'YY')` returns 01/01/1998 00:00:00, and `ROUND(07/1/1998 3:10:15, 'YY')` returns 1/1/1998 00:00:00.
- **Month.** Rounds the month portion of a date based on the day of the month. If the day of the month is between 1 and 15, it rounds the date to the first day of the input month with the time set to 00:00:00. If the day of the month is between 16 and the last day of the month, it rounds to the first day of the next month with the time set to 00:00:00. For example, the expression `ROUND(4/15/1998 12:15:00, 'MM')` returns 4/1/1998 00:00:00, and `ROUND(4/16/1998 8:24:19, 'MM')` returns 5/1/1998 00:00:00.
- **Day.** Rounds the day portion of the date based on the time. If the time is between 00:00:00 (12AM) and 11:59:59AM, the function returns the current date with the time set to 00:00:00 (12AM). If the time is 12:00:00 (12PM) or later, the function rounds the date to the next day with the time set to 00:00:00 (12AM). For example, the expression `ROUND(06/13/1998 2:30:45, 'DD')` returns 06/13/1998 00:00:00, and `ROUND(06/13/1998 22:30:45, 'DD')` returns 06/14/1998 00:00:00.
- **Hour.** Rounds the hour portion of the date based on the minutes in the hour. If the minute portion of the time is between 0 and 29, the function returns the current hour with the minutes and seconds set to 0. If the minute portion is 30 or greater, the function rounds to the next hour and sets the minutes and seconds to 0. For example, the expression `ROUND(04/01/1998 11:29:35, 'HH')` returns 04/01/1998 11:00:00, and `ROUND(04/01/1998 13:39:00, 'HH')` returns 04/01/1998 14:00:00.
- **Minute.** Rounds the minute portion of the date based on the seconds. If time has 0 to 29 seconds, the function returns the current minutes and sets the seconds to 0. If the time has 30 to 59 seconds, the function rounds to the next minute and sets the seconds to 0. For example, the expression `ROUND(05/22/1998 10:15:29, 'MI')` returns 05/22/1998 10:15:00, and `ROUND(05/22/1998 10:15:30, 'MI')` returns 05/22/1998 10:16:00.

## Syntax

```
ROUND( date [,format] )
```

| Argument | Required/ Optional | Description |
|---|---|---|
| *date* | Required | Date/Time datatype. You can nest TO_DATE to convert strings to dates before rounding. |
| *format* | Optional | Enter a valid format string. This is the portion of the date that you want to round. You can round only one portion of the date. If you omit the format string, the function rounds the date to the nearest day. |

```
Return Value
```

Date with the specified part rounded. ROUND returns a date in the same format as the source date. You can link the results of this function to any column with a Date/Time datatype.

NULL if you pass a null value to the function.

## Example

The following expressions round the year portion of dates in the DATE_SHIPPED column:

```
ROUND( DATE_SHIPPED, 'Y' )
ROUND( DATE_SHIPPED, 'YY' )
ROUND( DATE_SHIPPED, 'YYY' )
ROUND( DATE_SHIPPED, 'YYYY' )
```

| DATE_SHIPPED | RETURN VALUE |
|---|---|
| Jan 15 1998 2:10:30AM | Jan 1 1998 12:00:00AM |
| Apr 19 1998 1:31:20PM | Jan 1 1998 12:00:00AM |
| Dec 20 1998 3:29:55PM | Jan 1 1999 12:00:00AM |
| NULL | NULL |

The following expressions round the month portion of each date in the DATE_SHIPPED column:

```
ROUND( DATE_SHIPPED, 'MM' )
ROUND( DATE_SHIPPED, 'MON' )
ROUND( DATE_SHIPPED, 'MONTH' )
```

| DATE_SHIPPED | RETURN VALUE |
|---|---|
| Jan 15 1998 2:10:30AM | Jan 1 1998 12:00:00AM |
| Apr 19 1998 1:31:20PM | May 1 1998 12:00:00AM |
| Dec 20 1998 3:29:55PM | Jan 1 1999 12:00:00AM |
| NULL | NULL |

The following expressions round the day portion of each date in the DATE_SHIPPED column:

```
ROUND( DATE_SHIPPED, 'D' )
ROUND( DATE_SHIPPED, 'DD' )
ROUND( DATE_SHIPPED, 'DDD' )
```

```
    ROUND( DATE_SHIPPED, 'DY' )
    ROUND( DATE_SHIPPED, 'DAY' )
```

| DATE_SHIPPED | RETURN VALUE |
|---|---|
| Jan 15 1998 2:10:30AM | Jan 15 1998 12:00:00AM |
| Apr 19 1998 1:31:20PM | Apr 20 1998 12:00:00AM |
| Dec 20 1998 3:29:55PM | Dec 21 1998 12:00:00AM |
| Dec 31 1998 11:59:59PM | Jan 1 1999 12:00:00AM |
| NULL | NULL |

The following expressions round the hour portion of each date in the DATE_SHIPPED column:

```
    ROUND( DATE_SHIPPED, 'HH' )
    ROUND( DATE_SHIPPED, 'HH12' )
    ROUND( DATE_SHIPPED, 'HH24' )
```

| DATE_SHIPPED | RETURN VALUE |
|---|---|
| Jan 15 1998 2:10:31AM | Jan 15 1998 2:00:00AM |
| Apr 19 1998 1:31:20PM | Apr 19 1998 2:00:00PM |
| Dec 20 1998 3:29:55PM | Dec 20 1998 3:00:00PM |
| Dec 31 1998 11:59:59PM | Jan 1 1999 12:00:00AM |
| NULL | NULL |

The following expression rounds the minute portion of each date in the DATE_SHIPPED column:

```
    ROUND( DATE_SHIPPED, 'MI' )
```

| DATE_SHIPPED | RETURN VALUE |
|---|---|
| Jan 15 1998 2:10:30AM | Jan 15 1998 2:11:00AM |
| Apr 19 1998 1:31:20PM | Apr 19 1998 1:31:00PM |
| Dec 20 1998 3:29:55PM | Dec 20 1998 3:30:00PM |
| Dec 31 1998 11:59:59PM | Jan 1 1999 12:00:00AM |
| NULL | NULL |

# ROUND (Numbers)

Rounds numbers to a specified number of digits or decimal places. You can also use ROUND to round dates.

## Syntax

```
ROUND( numeric_value [, precision] )
```

| Argument | Required/ Optional | Description |
|---|---|---|
| *numeric_value* | Required | Numeric datatype. You can enter any valid expression. Use operators to perform arithmetic before you round the values. |
| *precision* | Optional | Positive or negative integer. If you enter a positive *precision*, the function rounds to this number of decimal places. For example, ROUND(12.99, 1) returns 13.0 and ROUND(15.44, 1) returns 15.4.<br>If you enter a negative *precision*, the function rounds this number of digits to the left of the decimal point, returning an integer. For example, ROUND(12.99, -1) returns 10 and ROUND(15.99, -1) returns 20.<br>If you enter decimal *precision*, the function rounds to the nearest integer before evaluating the expression. For example, ROUND(12.99, 0.8) returns 13.0 because the function rounds 0.8 to 1 and then evaluates the expression.<br>If you omit the *precision* argument, the function rounds to the nearest integer, truncating the decimal portion of the number. For example, ROUND(12.99) returns 13. |

## Return Value

Numeric value.

If one of the arguments is NULL, ROUND returns NULL.

## Example

The following expression returns the values in the Price column rounded to three decimal places.

```
ROUND( PRICE, 3 )
```

| PRICE | RETURN VALUE |
|---|---|
| 12.9936 | 12.994 |
| 15.9949 | 15.995 |
| -18.8678 | -18.868 |
| 56.9561 | 56.956 |
| NULL | NULL |

You can round digits to the left of the decimal point by passing a negative integer in the *precision* argument:

```
ROUND( PRICE, -2 )
```

| PRICE | RETURN VALUE |
|---|---|
| 13242.99 | 13200.0 |
| 1435.99 | 1400.0 |

| PRICE | RETURN VALUE |
|---|---|
| -108.95 | -100.0 |
| NULL | NULL |

If you pass a decimal value in the *precision* argument, Data Integration rounds it to the nearest integer before evaluating the expression:

```
ROUND( PRICE, 0.8 )
```

| PRICE | RETURN VALUE |
|---|---|
| 12.99 | 13.0 |
| 56.34 | 56.3 |
| NULL | NULL |

If you omit the *precision* argument, the function rounds to the nearest integer:

```
ROUND( PRICE )
```

| PRICE | RETURN VALUE |
|---|---|
| 12.99 | 13.0 |
| -15.99 | -16.0 |
| -18.99 | -19.0 |
| 56.95 | 57.0 |
| NULL | NULL |

## Tip

You can also use ROUND to explicitly set the precision of calculated values and achieve expected results.

# RPAD

Converts a string to a specified length by adding blanks or characters to the end of the string.

## Syntax

```
RPAD( first_string, length [,second_string] )
```

| Argument | Required/ Optional | Description |
|---|---|---|
| *first_string* | Required | Any string value. The strings you want to change. You can enter any valid expression. |
| *length* | Required | Must be a positive integer literal. Specifies the length you want each string to be. When *length* is a negative number, RPAD returns NULL. |
| *second_string* | Optional | Any string value. Passes the string you want to append to the right-side of the *first_string* values. Enclose the characters you want to add to the end of the string within single quotation marks, for example, 'abc'. This argument is case sensitive.<br>If you omit the second string, the function pads the end of the first string with blanks. |

## Return Value

String of the specified length.

NULL if a value passed to the function is NULL or if length is a negative number.

## Example

The following expression returns the item name with a length of 16 characters, appending the string '.' to the end of each item name:

```
RPAD( ITEM_NAME, 16, '.')
```

| ITEM_NAME | RETURN VALUE |
|---|---|
| Flashlight | Flashlight...... |
| Compass | Compass......... |
| Regulator System | Regulator System |
| Safety Knife | Safety Knife.... |

RPAD counts the length from left to right. So, if the first string is longer than the length, RPAD truncates the string from right to left. For example, RPAD('alphabetical', 5, 'x') would return the string 'alpha'. RPAD uses a partial part of the *second_string* when necessary.

The following expression returns the item name with a length of 16 characters, appending the string '*..*' to the end of each item name:

```
RPAD( ITEM_NAME, 16, '*..*' )
```

| ITEM_NAME | RETURN VALUE |
|---|---|
| Flashlight | Flashlight*..**. |
| Compass | Compass*..**..** |

| ITEM_NAME | RETURN VALUE |
|---|---|
| Regulator System | Regulator System |
| Safety Knife | Safety Knife*..* |

The following expression shows how RPAD handles negative values for the *length* argument for each row in the ITEM_NAME column:

```
RPAD( ITEM_NAME, -5, '.')
```

| ITEM_NAME | RETURN VALUE |
|---|---|
| Flashlight | NULL |
| Compass | NULL |
| Regulator System | NULL |

# RTRIM

Removes blanks or characters from the end of a string.

If you do not specify a *trim_set* parameter in the expression, RTRIM removes only single-byte spaces.

If you use RTRIM to remove characters from a string, RTRIM compares the *trim_set* to each character in the *string* argument, character-by-character, starting with the right side of the string. If the character in the string matches any character in the *trim_set*, RTRIM removes it. RTRIM continues comparing and removing characters until it fails to find a matching character in the *trim_set*. It returns the string without the matching characters.

## Syntax

```
RTRIM( string [, trim_set] )
```

| Argument | Required/ Optional | Description |
|---|---|---|
| *string* | Required | Any string value. Passes the values you want to trim. You can enter any valid expression. Use operators to perform comparisons or concatenate strings before removing blanks from the end of a string. |
| *trim_set* | Optional | Any string value. Passes the characters you want to remove from the end of the string. You can also enter a text literal. However, you must enclose the characters you want to remove from the end of the string within single quotation marks, for example, 'abc'. If you omit the second string, the function removes blanks from the end of the first string.<br>RTRIM is case sensitive. |

## Return Value

String. The string values with the specified characters in the *trim_set* argument removed.

NULL if a value passed to the function is NULL.

### Example

The following expression removes the characters ‘re’ from the strings in the LAST_NAME column:

```
RTRIM( LAST_NAME, 're')
```

| LAST_NAME | RETURN VALUE |
|---|---|
| Nelson | Nelson |
| Page | Pag |
| Osborne | Osborn |
| NULL | NULL |
| Sawyer | Sawy |
| H. Bender | H. Bend |
| Steadman | Steadman |

RTRIM removes ‘e’ from Page even though ‘r’ is the first character in the *trim_set*. This is because RTRIM searches, character-by-character, for the set of characters you specify in the *trim_set* argument. If the last character in the string matches the first character in the *trim_set*, RTRIM removes it. If, however, the last character in the string does not match, RTRIM compares the second character in the *trim_set*. If the second from last character in the string matches the second character in the *trim_set*, RTRIM removes it, and so on. When the character in the string fails to match the *trim_set*, RTRIM returns the string and evaluates the next row.

In the last example, the last character in Nelson does not match any character in the *trim_set* argument, so RTRIM returns the string ‘Nelson’ and evaluates the next row.

### Tips

Use RTRIM and LTRIM with || or CONCAT to remove leading and trailing blanks after you concatenate two strings.

You can also remove multiple sets of characters by nesting RTRIM. For example, if you want to remove trailing blanks and the character ‘t’ from the end of each string in a column of names, you might create an expression similar to the following:

```
RTRIM( RTRIM( NAMES ), 't' )
```

# SET_DATE_PART

Sets one part of a date/time value to a value you specify.

With SET_DATE_PART, you can change the following parts of a date:

**Year**

Change the year by entering a positive integer in the *value* argument. Use any of the year format strings: Y, YY, YYY, or YYYY to set the year. For example, the expression `SET_DATE_PART( SHIP_DATE, 'YY', 2001 )` changes the year to 2001 for all dates in the SHIP_DATE column.

**Month**

Change the month by entering a positive integer between 1 and 12 (January=1 and December=12) in the *value* argument. Use any of the month format strings: MM, MON, MONTH to set the month. For example, the expression `SET_DATE_PART( SHIP_DATE, 'MONTH', 10 )` changes the month to October for all dates in the SHIP_DATE column.

**Day**

Change the day by entering a positive integer between 1 and 31 (except for the months that have less than 31 days: February, April, June, September, and November) in the *value* argument. Use any of the month format strings (D, DD, DDD, DY, and DAY) to set the day. For example, the expression `SET_DATE_PART( SHIP_DATE, 'DD', 10 )` changes the day to 10 for all dates in the SHIP_DATE column.

**Hour**

Change the hour by entering a positive integer between 0 and 24 (where 0=12AM, 12=12PM, and 24 =12AM) in the *value* argument. Use any of the hour format strings (HH, HH12, HH24) to set the hour. For example, the expression `SET_DATE_PART( SHIP_DATE, 'HH', 14 )` changes the hour to 14:00:00 (or 2:00:00PM) for all dates in the SHIP_DATE column.

**Minute**

Change the minutes by entering a positive integer between 0 and 59 in the *value* argument. You use the MI format string to set the minute. For example, the expression `SET_DATE_PART( SHIP_DATE, 'MI', 25 )` changes the minute to 25 for all dates in the SHIP_DATE column.

**Seconds**

You can change the seconds by entering a positive integer between 0 and 59 in the *value* argument. You use the SS format string to set the second. For example, the expression `SET_DATE_PART( SHIP_DATE, 'SS', 59 )` changes the second to 59 for all dates in the SHIP_DATE column.

## Syntax

```
SET_DATE_PART( date, format, value )
```

| Argument | Required/ Optional | Description |
|---|---|---|
| *date* | Required | Date/Time datatype. The date you want to modify. You can enter any valid expression. |
| *format* | Required | A format string specifying the portion of the date to be changed. The format string is not case sensitive. |
| *value* | Required | A positive integer value assigned to the specified portion of the date. The integer must be a valid value for the part of the date you want to change. If you enter an improper value (for example, February 30), the session fails. |

## Return Value

Date in the same format as the source date with the specified part changed.

NULL if a value passed to the function is NULL.

## Example

The following expressions change the hour to 4PM for each date in the DATE_PROMISED column:

```
SET_DATE_PART( DATE_PROMISED, 'HH', 16 )
SET_DATE_PART( DATE_PROMISED, 'HH12', 16 )
SET_DATE_PART( DATE_PROMISED, 'HH24', 16 )
```

| DATE_PROMISED | RETURN VALUE |
|---|---|
| Jan 1 1997 12:15:56AM | Jan 1 1997 4:15:56PM |
| Feb 13 1997 2:30:01AM | Feb 13 1997 4:30:01PM |
| Mar 31 1997 5:10:15PM | Mar 31 1997 4:10:15PM |
| Dec 12 1997 8:07:33AM | Dec 12 1997 4:07:33PM |
| NULL | NULL |

The following expressions change the month to June for the dates in the DATE_PROMISED column. Data Integration displays an error when you try to create a date that does not exist, such as changing March 31 to June 31:

```
SET_DATE_PART( DATE_PROMISED, 'MM', 6 )
SET_DATE_PART( DATE_PROMISED, 'MON', 6 )
SET_DATE_PART( DATE_PROMISED, 'MONTH', 6 )
```

| DATE_PROMISED | RETURN VALUE |
|---|---|
| Jan 1 1997 12:15:56AM | Jun 1 1997 12:15:56AM |
| Feb 13 1997 2:30:01AM | Jun 13 1997 2:30:01AM |
| Mar 31 1997 5:10:15PM | None. Data Integration writes the row into the error rows file. |
| Dec 12 1997 8:07:33AM | Jun 12 1997 8:07:33AM |
| NULL | NULL |

The following expressions change the year to 2000 for the dates in the DATE_PROMISED column:

```
SET_DATE_PART( DATE_PROMISED, 'Y', 2000 )
SET_DATE_PART( DATE_PROMISED, 'YY', 2000 )
SET_DATE_PART( DATE_PROMISED, 'YYY', 2000 )
SET_DATE_PART( DATE_PROMISED, 'YYYY', 2000 )
```

| DATE_PROMISED | RETURN VALUE |
|---|---|
| Jan 1 1997 12:15:56AM | Jan 1 2000 12:15:56AM |
| Feb 13 1997 2:30:01AM | Feb 13 2000 2:30:01AM |
| Mar 31 1997 5:10:15PM | Mar 31 2000 5:10:15PM |
| Dec 12 1997 8:07:33AM | Dec 12 2000 4:07:33PM |
| NULL | NULL |

### Tip

If you want to change multiple parts of a date at one time, you can nest multiple SET_DATE_PART functions within the *date* argument. For example, you might write the following expression to change all of the dates in the DATE_ENTERED column to July 1 1998:

```
SET_DATE_PART( SET_DATE_PART( SET_DATE_PART( DATE_ENTERED, 'YYYY', 1998),MM', 7), 'DD',
1)
```

# SETCOUNTVARIABLE

Counts the rows evaluated by the function and increments the current value of an in-out parameter based on the count. Returns the new current value.

At the end of each task session, Data Integration saves the last current value in the job details. Unless overridden, Data Integration uses the saved value as the initial value of the variable for the next time you use the task.

Use the SETCOUNTVARIABLE function only once for each in-out parameter in a pipeline. Data Integration processes variable functions as it encounters them in the mapping. The order in which Data Integration encounters variable functions in the mapping might not be the same for every session run. This could cause inconsistent results when you use the same variable function multiple times in a mapping.

You can use SETCOUNTVARIABLE in an Expression transformation.

## Syntax

```
SETCOUNTVARIABLE( $$Variable )
```

| Argument | Required/ Optional | Description |
|---|---|---|
| *$$Variable* | Required | Name of the in-out parameter that you want to set. |

## Return Value

The current value of the in-out parameter.

# SETMAXVARIABLE

Sets the current value of an in-out parameter to the higher of two values: the current value of the parameter or the value you specify. Returns the new current value.

At the end of a task session, Data Integration saves the final current value and writes it to the job details. Unless overridden, it uses the saved value as the initial value of the parameter for the next task session.

When used with a string in-out parameter, SETMAXVARIABLE returns the higher string based on the sort order selected in the mapping.

Use the SETMAXVARIABLE function only once for each in-out parameter in a pipeline. Data Integration processes in-out parameters as it encounters them in a mapping. The order in which Data Integration

encounters variable functions in the mapping might not be the same for every task session. This could cause inconsistent results if you use the same variable function multiple times in a mapping.

You can use SETMAXVARIABLE with the Expression transformation.

### Syntax

```
SETMAXVARIABLE( $$Variable, value )
```

| Argument | Required/ Optional | Description |
|---|---|---|
| $$ *Variable* | Required | Name of the in-out parameter you want to set. Use in-out parameters with Max aggregation type. |
| *value* | Required | The value you want Data Integration to compare against the current value of the in-out parameter. You can enter any valid transformation expression that evaluates to a data type compatible with the data type of the parameter. |

### Return Value

The higher of two values: the current value of the in-out parameter or the value you specified. The return value is the new current value of the parameter.

When *value* is NULL Data Integration returns the current value of *$$Variable*.

# SETMINVARIABLE

Sets the current value of an in-out parameter to the lower of two values: the current value of the parameter or the value you specify. Returns the new current value.

Data Integration saves the final current value in the job details. Unless overridden, it uses the saved value as the initial value of the parameter for the next task session.

When used with a string in-out parameter, SETMINVARIABLE returns the lower string based on the sort order selected for the mapping.

Use the SETMINVARIABLE function only once for each in-out parameter in a pipeline. Data Integration processes variable functions as it encounters them in the mapping. The order in which Data Integration encounters variable functions in the mapping might not be the same for every session. This could cause inconsistent results if you use the same variable function multiple times in a mapping.

You can use SETMINVARIABLE in the Expression transformation.

### Syntax

```
SETMINVARIABLE( $$Variable, value )
```

| Argument | Required/ Optional | Description |
|---|---|---|
| $$ *Variable* | Required | Name of the in-out parameter you want to set. Use with in-out parameter with Min aggregation type. |
| *value* | Required | The value you want Data Integration to compare against the current value of the parameter. You can enter any valid expression that evaluates to a data type compatible with the data type of the parameter. |

### Return Value

The lower of two values: the current value of the parameter or the value you specified. The return value is the new current value of the parameter.

When *value* is NULL, Data Integration returns the current value of *$$Variable*.

# SETVARIABLE

Sets the current value of an in-out parameter to a value you specify. Returns the specified value.

During task execution, Data Integration compares the final current value of the in-out parameter to the start value. Based on the aggregate type of the in-out parameter, it saves a final current value in the job details. Unless overridden, it uses the saved value as the initial value of the in-out parameter for the next task run.

Use the SETVARIABLE function only once for each in-out parameter in a pipeline. Data Integration processes variable functions as it encounters them in the mapping. The order in which Data Integration encounters variable functions in the mapping may not be the same for every task session. This could cause inconsistent results if you use the same variable function multiple times in a mapping.

You can use SETVARIABLE in the Expression transformation.

### Syntax

```
SETVARIABLE( $$Variable, value )
```

| Argument | Required/ Optional | Description |
|---|---|---|
| $$ *Variable* | Required | Name of the in-out parameter you want to set. Use with in-out parameters configured with the Max or Min aggregation type. |
| *value* | Required | The value to set as the current value of the in-out parameter. You can use a valid expression that evaluates to a datatype compatible with the datatype of the parameter. |

### Return Value

Current value of the in-out parameter.

When *value* is NULL, Data Integration returns the current value of *$$Variable*.

### Examples

The following expression sets an in-out parameter $$Time to the system date at the time Data Integration evaluates the row and returns the system date to the SET_$$TIME port:

```
SETVARIABLE ($$Time, SYSDATE)
```

| TRANSACTION | TOTAL | SET_$$TIME |
|---|---|---|
| 0100002 | 534.23 | 10/10/2016 01:34:33 |
| 0100003 | 699.01 | 10/10/2016 01:34:34 |
| 0100004 | 97.50 | 10/10/2016 01:34:35 |
| 0100005 | 116.43 | 10/10/2016 01:34:36 |
| 0100006 | 323.95 | 10/10/2016 01:34:37 |

At the end of the session, Data Integration saves 10/10/2016 01:34:37 in the job details as the last evaluated current value for $$Time. The next time the task runs, Data Integration evaluates all references to $$Time to 10/10/2016 01:34:37.

The following expression sets the in-out parameter $$Timestamp to the timestamp associated with the row and returns the timestamp to the SET_$$TIMESTAMP port:

```
SETVARIABLE ($$Time, TIMESTAMP)
```

| TRANSACTION | TIMESTAMP | TOTAL | SET_$$TIME |
|---|---|---|---|
| 0100002 | 10/01/2016 12:01:01 | 534.23 | 10/01/2016 12:01:01 |
| 0100003 | 10/01/2016 12:10:22 | 699.01 | 10/01/2016 12:10:22 |
| 0100004 | 10/01/2016 12:16:45 | 97.50 | 10/01/2016 12:16:45 |
| 0100005 | 10/01/2016 12:23:10 | 116.43 | 10/01/2016 12:23:10 |
| 0100006 | 10/01/2016 12:40:31 | 323.95 | 10/01/2016 12:40:31 |

At the end of the session, Data Integration saves 10/01/2016 12:40:31 in the job details as the last evaluated current value for $$Timestamp.

The next time the session runs, Data Integration sets the initial value of $$Timestamp to 10/01/2016 12:40:31.

# SHA256

Returns the SHA-256 digest of the input value. The function uses Secure Hash Algorithm 2 (SHA-2) and returns a 256 bit digest. SHA256 is a one-way cryptographic hash function with a 256-bit hash value. You can conclude that input values are different when the SHA-256 digests of the input values are different.

Use SHA256 to verify data integrity or to generate unique keys.

SHA256 returns different values when the connection reads ASCII and Unicode data.

### Syntax

```
SHA256( value )
```

The following table describes the argument for this command:

| Argument | Required/Optional | Description |
|---|---|---|
| value | Required | String or binary datatype. The case-sensitive value for which you want to calculate digest. |

### Return value

Unique 32-byte binary.

NULL if the input is a null value.

### Example

You want to write changed data to a database. Use SHA256 to generate hash digest values for rows of data that Data Integration reads from a source. When you run the task, compare the previously generated checksum values to the new checksum values. You can conclude that an updated checksum value indicates that the data has changed. You write rows with updated checksum values to the target.

## SIGN

Returns whether a numeric value is positive, negative, or 0.

### Syntax

```
SIGN( numeric_value )
```

| Argument | Required/ Optional | Description |
|---|---|---|
| *numeric_value* | Required | Numeric value. Passes the values you want to evaluate. You can enter any valid expression. |

### Return Value

-1 for negative values.

0 for 0.

1 for positive values.

NULL if NULL.

### Example

The following expression determines if the SALES column includes any negative values:

```
SIGN( SALES )
```

| SALES | RETURN VALUE |
|---|---|
| 100 | 1 |

| SALES | RETURN VALUE |
|---|---|
| -25.99 | -1 |
| 0 | 0 |
| NULL | NULL |

# SIN

Returns the sine of a numeric value (expressed in radians).

## Syntax

```
SIN( numeric_value )
```

| Argument | Required/ Optional | Description |
|---|---|---|
| *numeric_value* | Required | Numeric datatype. Numeric data expressed in radians (degrees multiplied by pi divided by 180). Passes the values for which you want to calculate the sine. You can enter any valid expression. You can also use operators to convert a numeric value to radians or perform arithmetic within the SIN calculation. |

## Return Value

Double value.

NULL if a value passed to the function is NULL.

## Example

The following expression converts the values in the Degrees column to radians and then calculates the sine for each radian:

```
SIN( DEGREES * 3.14159265359 / 180 )
```

| DEGREES | RETURN VALUE |
|---|---|
| 0 | 0 |
| 90 | 1 |
| 70 | 0.939692620785936 |
| 30 | 0.50000000000003 |
| 5 | 0.0871557427476639 |
| 18 | 0.309016994374967 |
| 89 | 0.999847695156393 |
| NULL | NULL |

You can perform arithmetic on the values passed to SIN before the function calculates the sine. For example:

```
SIN( ARCS * 3.14159265359 / 180 )
```

# SINH

Returns the hyperbolic sine of the numeric value.

## Syntax

```
SINH( numeric_value )
```

| Argument | Required/ Optional | Description |
|---|---|---|
| *numeric_value* | Required | Numeric datatype. Numeric data expressed in radians (degrees multiplied by pi divided by 180). Passes the values for which you want to calculate the hyperbolic sine. You can enter any valid expression. |

## Return Value

Double value.

NULL if a value passed to the function is NULL.

## Example

The following expression returns the hyperbolic sine for the values in the ANGLES column:

```
SINH( ANGLES )
```

| ANGLES | RETURN VALUE |
|---|---|
| 1.0 | 1.1752011936438 |
| 2.897 | 9.03225804884884 |
| 3.66 | 19.4178051793031 |
| 5.45 | 116.376934801486 |
| 0 | 0.0 |
| 0.345 | 0.35188478309993 |
| NULL | NULL |

## Tip

You can perform arithmetic on the values passed to SINH before the function calculates the hyperbolic sine. For example:

```
SINH( MEASURES.ARCS / 180 )
```

# SOUNDEX

Encodes a string value into a four-character string.

SOUNDEX works for characters in the English alphabet (A-Z). It uses the first character of the input string as the first character in the return value and encodes the remaining three unique consonants as numbers.

SOUNDEX encodes characters according to the following list of rules:

- Uses the first character in *string* as the first character in the return value and encodes it in uppercase. For example, both SOUNDEX('John') and SOUNDEX('john') return 'J500'.
- Encodes the first three unique consonants following the first character in *string* and ignores the rest. For example, both SOUNDEX('JohnRB') and SOUNDEX('JohnRBCD') return 'J561'.
- Assigns a single code to consonants that sound alike.
  The following table lists SOUNDEX encoding guidelines for consonants:

| Code | Consonant |
|---|---|
| 1 | B, P, F, V |
| 2 | C, S, G, J, K, Q, X, Z |
| 3 | D, T |
| 4 | L |
| 5 | M, N |
| 6 | R |

- Skips the characters A, E, I, O, U, H, and W unless one of them is the first character in *string*. For example, SOUNDEX('A123') returns 'A000' and SOUNDEX('MAeiouhwC') returns 'M000'.
- If *string* produces fewer than four characters, SOUNDEX pads the resulting string with zeroes. For example, SOUNDEX('J') returns 'J000'.
- If *string* contains a set of consecutive consonants that use the same code listed in the table above, SOUNDEX encodes the first occurrence and skips the remaining occurrences in the set. For example, SOUNDEX('AbbpdMN') returns 'A135'.
- Skips numbers in *string*. For example, both SOUNDEX('Joh12n') and SOUNDEX('1John') return 'J500'.
- Returns NULL if *string* is NULL or if all the characters in *string* are not letters of the English alphabet.

## Syntax

```
SOUNDEX( string )
```

| Argument | Required/ Optional | Description |
|---|---|---|
| *string* | Required | Character string. Passes the string value you want to encode. You can enter any valid expression. |

## Return Value

String.

NULL if one of the following conditions is true:

- If value passed to the function is NULL.
- No character in *string* is a letter of the English alphabet.
- *string* is empty.

### Example

The following expression encodes the values in the EMPLOYEE_NAME column:

```
SOUNDEX( EMPLOYEE_NAME )SOUNDEX
```

| EMPLOYEE_NAME | RETURN VALUE |
|---|---|
| John | J500 |
| William | W450 |
| jane | J500 |
| joh12n | J500 |
| 1abc | A120 |
| NULL | NULL |

# SQRT

Returns the square root of a non-negative numeric value.

### Syntax

```
SQRT( numeric_value )
```

| Argument | Required/ Optional | Description |
|---|---|---|
| *numeric_value* | Required | Positive numeric value. Passes the values for which you want to calculate a square root. You can enter any valid expression. |

### Return Value

Double value.

NULL if a value passed to the function is NULL.

### Example

The following expression returns the square root for the values in the NUMBERS column:

```
SQRT( NUMBERS )
```

| NUMBERS | RETURN VALUE |
|---|---|
| 100 | 10 |
| -100 | None. Data Integration writes the row into the error rows file. |

```
NUMBERS          RETURN VALUE

NULL             NULL

60.54            7.78074546557076
```

The value -100 results in an error during the session, since the function SQRT only evaluates positive numeric values. If you pass a negative value or character value, Data Integration writes the row into the error rows file.

You can perform arithmetic on the values passed to SQRT before the function calculates the square root.

# STDDEV

Returns the standard deviation of the numeric values you pass to this function. STDDEV is used to analyze statistical data.

You can nest only one other aggregate function within STDDEV, and the nested function must return a numeric data type. You cannot nest aggregate functions in advanced mode.

Use only in mapping tasks.

## Syntax

```
STDDEV( numeric_value [,filter_condition] )
```

| Argument | Required/ Optional | Description |
|---|---|---|
| *numeric_value* | Required | Numeric data types. This function passes the values for which you want to calculate a standard deviation or the results of a function. You can enter any valid transformation expression. You can use operators to average values in different fields. |
| *filter_condition* | Optional | Limits the rows in the search. The filter condition must be a numeric value or evaluate to TRUE, FALSE, or NULL. You can enter any valid transformation expression. |

## Return Value

Numeric value.

NULL if all values passed to the function are NULL or if no rows are selected (for example, the filter condition evaluates to FALSE or NULL for all rows).

If you use STDDEV in an Aggregator transformation with a group by field that has only one row, the Data Integration Server returns a standard deviation of 0 while an advanced cluster returns NULL.

## Nulls

If a single value is NULL, STDDEV ignores it. However, if all values are NULL, STDDEV returns NULL.

## Group By

STDDEV groups values based on group by fields you define in the transformation, returning one result for each group.

If there is no group by field, STDDEV treats all rows as one group, returning one value.

## Examples

The following expression calculates the standard deviation of all rows greater than $2000.00 in the TOTAL_SALES field:

```
STDDEV( SALES, SALES > 2000.00 )
```

| SALES |
| --- |
| 2198.0 |
| 1010.90 |
| 2256.0 |
| 153.88 |
| 3001.0 |
| NULL |
| 8953.0 |

**RETURN VALUE:** 3254.60361129688

The function does not include the values 1010.90 and 153.88 in the calculation because the *filter_condition* specifies sales greater than $2,000.

The following expression calculates the standard deviation of all rows in the SALES field:

```
STDDEV(SALES)
```

| SALES |
| --- |
| 2198.0 |
| 2198.0 |
| 2198.0 |
| 2198.0 |

**RETURN VALUE:** 0

The return value is 0 because each row contains the same number (no standard deviation exists). If there is no standard deviation, the return value is 0.

# SUBSTR

Returns a portion of a string. SUBSTR counts all characters, including blanks, starting at the beginning of the string.

## Syntax

```
SUBSTR( string, start [,length] )
```

| Argument | Required/ Optional | Description |
|---|---|---|
| *string* | Required | Must be a character string. Passes the strings you want to search. You can enter any valid expression. If you pass a numeric value, the function converts it to a character string. |
| *start* | Required | Must be an integer. The position in the string where you want to start counting. You can enter any valid expression. If the start position is a positive number, SUBSTR locates the start position by counting from the beginning of the string. If the start position is a negative number, SUBSTR locates the start position by counting from the end of the string. If the start position is 0, SUBSTR searches from the first character in the string. |
| *length* | Optional | Must be an integer greater than 0. The number of characters you want SUBSTR to return. You can enter any valid expression. If you omit the length argument, SUBSTR returns all of the characters from the start position to the end of the string. If you pass a negative integer or 0, the function returns an empty string. If you pass a decimal, the function rounds it to the nearest integer value. |

## Return Value

String.

Empty string if you pass a negative or 0 length value.

NULL if a value passed to the function is NULL.

## Example

The following expressions return the area code for each row in the PHONE column:

```
SUBSTR( PHONE, 0, 3 )
```

| PHONE | RETURN VALUE |
|---|---|
| 809-555-0269 | 809 |
| 357-687-6708 | 357 |
| NULL | NULL |

```
SUBSTR( PHONE, 1, 3 )
```

| PHONE | RETURN VALUE |
|---|---|
| 809-555-3915 | 809 |
| 357-687-6708 | 357 |
| NULL | NULL |

The following expressions return the phone number without the area code for each row in the PHONE column:

```
SUBSTR( PHONE, 5, 8 )
```

| PHONE | RETURN VALUE |
|---|---|
| 808-555-0269 | 555-0269 |
| 809-555-3915 | 555-3915 |
| 357-687-6708 | 687-6708 |
| NULL | NULL |

You can also pass a negative start value to return the phone number for each row in the PHONE column. The expression still reads the source string from left to right when returning the result of the *length* argument:

```
SUBSTR( PHONE, -8, 3 )
```

| PHONE | RETURN VALUE |
|---|---|
| 808-555-0269 | 555 |
| 809-555-3915 | 555 |
| 357-687-6708 | 687 |
| NULL | NULL |

You can nest INSTR in the *start* or *length* argument to search for a specific string and return its position.

The following expression evaluates a string, starting from the end of the string. The expression finds the last (right-most) space in the string and then returns all characters preceding it:

```
SUBSTR( CUST_NAME,1,INSTR( CUST_NAME,' ' ,-1,1 ) - 1 )
```

| CUST_NAME | RETURN VALUE |
|---|---|
| PATRICIA JONES | PATRICIA |
| MARY ELLEN SHAH | MARY ELLEN |

The following expression removes the character '#' from a string:

```
SUBSTR( CUST_ID, 1, INSTR(CUST_ID, '#')-1 ) || SUBSTR( CUST_ID, INSTR(CUST_ID, '#')+1 )
```

When the *length* argument is longer than the string, SUBSTR returns all the characters from the start position to the end of the string. Consider the following example:

```
SUBSTR('abcd', 2, 8)
```

The return value is ‘bcd’. Compare this result to the following example:

```
SUBSTR('abcd', -2, 8)
```

The return value is ‘cd’.

# SUM

Returns the sum of all values in the selected field. Optionally, you can apply a filter to limit the rows you read to calculate the total.

You can nest only one other aggregate function within SUM, and the nested function must return a numeric data type. You cannot nest aggregate functions in advanced mode.

Use only in mapping tasks.

## Syntax

```
SUM( numeric_value [, filter_condition ] )
```

| Argument | Required/ Optional | Description |
|---|---|---|
| *numeric_value* | Required | Numeric data type. Passes the values you want to add. You can enter any valid transformation expression. You can use operators to add values in different fields. |
| *filter_condition* | Optional | Limits the rows in the search. The filter condition must be a numeric value or evaluate to TRUE, FALSE, or NULL. You can enter any valid transformation expression. |

## Return Value

Numeric value.

NULL if all values passed to the function are NULL or if no rows are selected (for example, the filter condition evaluates to FALSE or NULL for all rows).

## Nulls

If a single value is NULL, SUM ignores it. However, if all values passed from the field are NULL, SUM returns NULL.

## Group By

SUM groups values based on group by fields you define in the transformation, returning one result for each group.

If there is no group by field, SUM treats all rows as one group, returning one value.

## Example

The following expression returns the sum of all values greater than 2000 in the Sales field:

```
SUM( SALES, SALES > 2000 )
```

| SALES |
|---|
| 2500.0 |
| 1900.0 |
| 1200.0 |
| NULL |
| 3458.0 |

```
SALES

4519.0

RETURN VALUE: 10477.0
```

### Tip

You can perform arithmetic on the values passed to SUM before the function calculates the total. For example:

```
SUM( QTY * PRICE - DISCOUNT )
```

# SYSTIMESTAMP

Returns the current date and time with precision to the nanosecond of the system that hosts the Secure Agent that starts the task. The precision to which you can retrieve the date and time depends on the system that hosts the Secure Agent.

The return value of the function varies depending on how you configure the argument:

- When you configure the argument of SYSTIMESTAMP as a variable, Data Integration evaluates the function for each row in the transformation.
- When you configure the argument of SYSTIMESTAMP as a constant, Data Integration evaluates the function once and retains the value for each row in the transformation.

## Syntax

```
SYSTIMESTAMP( [format] )
```

| Argument | Required/ Optional | Description |
|---|---|---|
| format | Optional | Precision to which you want to retrieve the timestamp. You can specify precision up to seconds (SS), milliseconds (MS), microseconds (US), or nanoseconds (NS). Enclose the format string within single quotation marks. The format string is not case sensitive. For example, to display the date and time to the precision of milliseconds use the following syntax: SYSTIMESTAMP('MS'). Default precision is microseconds (US). |

## Return Value

Timestamp. Returns date and time to the specified precision. Precision dependent on platform.

## Examples

Your organization has an online order service and processes real-time data. You can use the SYSTIMESTAMP function to generate a primary key for each transaction in the target database.

Create an Expression transformation with the following fields and values:

```
Field Name              Field Type          Expression

Customer_Name           Input               n/a

Order_Qty               Input               n/a
```

| Field Name | Field Type | Expression |
|---|---|---|
| Time_Counter | Variable | 'US' |
| Transaction_ID | Output | SYSTIMESTAMP (Time_Counter) |

At run time, the SYSTIMESTAMP generates the system time to the precision of microseconds for each row:

| Customer_Name | Order_Qty | Transaction_Id |
|---|---|---|
| Vani Deed | 14 | 07/06/2007 18:00:30.701015000 |
| Kalia Crop | 3 | 07/06/2007 18:00:30.701029000 |
| Vani Deed | 6 | 07/06/2007 18:00:30.701039000 |
| Harry Spoon | 32 | 07/06/2007 18:00:30.701048000 |

# TAN

Returns the tangent of a numeric value (expressed in radians).

## Syntax

```
TAN( numeric_value )
```

| Argument | Required/ Optional | Description |
|---|---|---|
| numeric_value | Required | Numeric datatype. Numeric data expressed in radians (degrees multiplied by pi divided by 180). Passes the numeric values for which you want to calculate the tangent. You can enter any valid expression. |

## Return Value

Double value.

NULL if a value passed to the function is NULL.

## Example

The following expression returns the tangent for all values in the DEGREES column:

```
TAN( DEGREES * 3.14159 / 180 )
```

| DEGREES | RETURN VALUE |
|---|---|
| 70 | 2.74747741945531 |
| 50 | 1.19175359259435 |
| 30 | 0.577350269189672 |
| 5 | 0.0874886635259298 |
| 18 | 0.324919696232929 |

| DEGREES | RETURN VALUE |
|---|---|
| 89 | 57.2899616310952 |
| NULL | NULL |

# TANH

Returns the hyperbolic tangent of the numeric value passed to this function.

## Syntax

```
TANH( numeric_value )
```

| Argument | Required/ Optional | Description |
|---|---|---|
| *numeric_value* | Required | Numeric datatype. Numeric data expressed in radians (degrees multiplied by pi divided by 180). Passes the numeric values for which you want to calculate the hyperbolic tangent. You can enter any valid expression. |

## Return Value

Double value.

NULL if a value passed to the function is NULL.

## Example

The following expression returns the hyperbolic tangent for the values in the ANGLES column:

```
TANH( ANGLES )
```

| ANGLES | RETURN VALUE |
|---|---|
| 1.0 | 0.761594155955765 |
| 2.897 | 0.993926947790665 |
| 3.66 | 0.998676551914886 |
| 5.45 | 0.999963084213409 |
| 0 | 0.0 |
| 0.345 | 0.331933853503641 |
| NULL | NULL |

## Tip

You can perform arithmetic on the values passed to TANH before the function calculates the hyperbolic tangent. For example:

```
TANH( ARCS / 360 )
```

# TO_BIGINT

Converts a string or numeric value to a bigint value. TO_BIGINT syntax contains an optional argument that you can choose to round the number to the nearest integer or truncate the decimal portion. TO_BIGINT ignores leading blanks.

## Syntax

```
TO_BIGINT( value [, flag] )
```

| Argument | Required/ Optional | Description |
|---|---|---|
| *value* | Required | String or numeric datatype. Passes the value you want to convert to a bigint value. You can enter any valid expression. |
| *flag* | Optional | Specifies whether to truncate or round the decimal portion. The flag must be an integer literal or the constants TRUE or FALSE:<br>- TO_BIGINT truncates the decimal portion when the flag is TRUE or a number other than 0.<br>- TO_BIGINT rounds the value to the nearest integer if the flag is FALSE or 0 or if you omit this argument.<br>The flag is not set by default. |

## Return Value

Bigint.

NULL if the value passed to the function is NULL.

0 if the value passed to the function contains alphanumeric characters.

## Examples

The following expressions use values from the IN_TAX source column:

```
TO_BIGINT( IN_TAX, TRUE )
```

| IN_TAX | RETURN VALUE |
|---|---|
| '7,245,176,201,123,435.6789' | 7,245,176,201,123,435 |
| '7,245,176,201,123,435.2' | 7,245,176,201,123,435 |
| '7,245,176,201,123,435.2.48' | 7,245,176,201,123,435 |
| NULL | NULL |
| 'A12.3Grove' | 0 |
| '  176,201,123,435.87' | 176,201,123,435 |
| '-7,245,176,201,123,435.2’ | -7,245,176,201,123,435 |
| '-7,245,176,201,123,435.23' | -7,245,176,201,123,435 |
| -9,223,372,036,854,775,806.9 | -9,223,372,036,854,775,806 |

| IN_TAX | RETURN VALUE |
|---|---|
| 9,223,372,036,854,775,806.9 | 9,223,372,036,854,775,806 |

```
    TO_BIGINT( IN_TAX )
```

| IN_TAX | RETURN VALUE |
|---|---|
| '7,245,176,201,123,435.6789' | 7,245,176,201,123,436 |
| '7,245,176,201,123,435.2' | 7,245,176,201,123,435 |
| '7,245,176,201,123,435.348' | 7,245,176,201,123,435 |
| NULL | NULL |
| 'A12.3Grove' | 0 |
| '    176,201,123,435.87' | 176,201,123,436 |
| '-7,245,176,201,123,435.6789' | -7,245,176,201,123,436 |
| '-7,245,176,201,123,435.23' | -7,245,176,201,123,435 |
| -9,223,372,036,854,775,806.9 | -9,223,372,036,854,775,807 |
| 9,223,372,036,854,775,806.9 | 9,223,372,036,854,775,807 |

# TO_CHAR (Dates)

Converts dates to character strings. TO_CHAR also converts numeric values to strings. You can convert the date into any format using the TO_CHAR format strings.

## Syntax

```
    TO_CHAR( date [,format] )
```

| Argument | Required/ Optional | Description |
|---|---|---|
| *date* | Required | Date/Time datatype. Passes the date values you want to convert to character strings. You can enter any valid expression. |
| *format* | Optional | Enter a valid TO_CHAR format string. The format string defines the format of the return value, not the format for the values in the date argument. If you omit the format string, the function returns a string based on the default date format of MM/DD/YYYY HH24:MI:SS. |

## Return Value

String.

NULL if a value passed to the function is NULL.

## Example

The following expression converts the dates in the DATE_PROMISED column to text in the format MON DD YYYY:

```
TO_CHAR( DATE_PROMISED, 'MON DD YYYY' )
```

| DATE_PROMISED | RETURN VALUE |
|---|---|
| Apr 1 1998 12:00:10AM | 'Apr 01 1998' |
| Feb 22 1998 01:31:10PM | 'Feb 22 1998' |
| Oct 24 1998 02:12:30PM | 'Oct 24 1998' |
| NULL | NULL |

If you omit the *format_string* argument, TO_CHAR returns a string in the default date format:

```
TO_CHAR( DATE_PROMISED )
```

| DATE_PROMISED | RETURN VALUE |
|---|---|
| Apr 1 1998 12:00:10AM | '04/01/1997 00:00:01' |
| Feb 22 1998 01:31:10PM | '02/22/1997 13:31:10' |
| Oct 24 1998 02:12:30PM | '10/24/1997 14:12:30' |
| NULL | NULL |

The following expressions return the day of the week for each date in a column:

```
TO_CHAR( DATE_PROMISED, 'D' )
```

| DATE_PROMISED | RETURN VALUE |
|---|---|
| 04-01-1997 12:00:10AM | '3' |
| 02-22-1997 01:31:10PM | '7' |
| 10-24-1997 02:12:30PM | '6' |
| NULL | NULL |

```
TO_CHAR( DATE_PROMISED, 'DAY' )
```

| DATE_PROMISED | RETURN VALUE |
|---|---|
| 04-01-1997 12:00:10AM | 'Tuesday' |
| 02-22-1997 01:31:10PM | 'Saturday' |
| 10-24-1997 02:12:30PM | 'Friday' |
| NULL | NULL |

The following expression returns the day of the month for each date in a column:

```
TO_CHAR( DATE_PROMISED, 'DD' )
```

| DATE_PROMISED | RETURN VALUE |
|---|---|
| 04-01-1997 12:00:10AM | '01' |
| 02-22-1997 01:31:10PM | '22' |
| 10-24-1997 02:12:30PM | '24' |
| NULL | NULL |

The following expression returns the day of the year for each date in a column:

```
TO_CHAR( DATE_PROMISED, 'DDD' )
```

| DATE_PROMISED | RETURN VALUE |
|---|---|
| 04-01-1997 12:00:10AM | '091' |
| 02-22-1997 01:31:10PM | '053' |
| 10-24-1997 02:12:30PM | '297' |
| NULL | NULL |

The following expressions return the hour of the day for each date in a column:

```
TO_CHAR( DATE_PROMISED, 'HH' )
TO_CHAR( DATE_PROMISED, 'HH12' )
```

| DATE_PROMISED | RETURN VALUE |
|---|---|
| 04-01-1997 12:00:10AM | '12' |
| 02-22-1997 01:31:10PM | '01' |
| 10-24-1997 02:12:30PM | '02' |
| NULL | NULL |

```
TO_CHAR( DATE_PROMISED, 'HH24' )
```

| DATE_PROMISED | RETURN VALUE |
|---|---|
| 04-01-1997 12:00:10AM | '00' |
| 02-22-1997 01:31:10PM | '13' |
| 10-24-1997 11:12:30PM | '23' |
| NULL | NULL |

The following expression converts date values to MJD values expressed as strings:

```
TO_CHAR( SHIP_DATE, 'J')
```

| SHIP_DATE | RETURN_VALUE |
|---|---|
| Dec 31 1999 03:59:59PM | 2451544 |
| Jan 1 1900 01:02:03AM | 2415021 |

The following expression converts dates to strings in the format MM/DD/YY:

```
TO_CHAR( SHIP_DATE, 'MM/DD/RR')
```

| SHIP_DATE | RETURN_VALUE |
|---|---|
| 12/31/1999 01:02:03AM | 12/31/99 |
| 09/15/1996 03:59:59PM | 09/15/96 |
| 05/17/2003 12:13:14AM | 05/17/03 |

You can also use the format string SSSSS in a TO_CHAR expression. For example, the following expression converts the dates in the SHIP_DATE column to strings representing the total seconds since midnight:

```
TO_CHAR( SHIP_DATE, 'SSSSS')
```

| SHIP_DATE | RETURN_VALUE |
|---|---|
| 12/31/1999 01:02:03AM | 3783 |
| 09/15/1996 03:59:59PM | 86399 |

In TO_CHAR expressions, the YY format string produces the same results as the RR format string.

The following expression converts dates to strings in the format MM/DD/YY:

```
TO_CHAR( SHIP_DATE, 'MM/DD/YY')
```

| SHIP_DATE | RETURN_VALUE |
|---|---|
| 12/31/1999 01:02:03AM | 12/31/99 |
| 09/15/1996 03:59:59PM | 09/15/96 |
| 05/17/2003 12:13:14AM | 05/17/03 |

The following expression returns the week of the month for each date in a column:

```
TO_CHAR( DATE_PROMISED, 'W' )
```

| DATE_PROMISED | RETURN VALUE |
|---|---|
| 04-01-1997 12:00:10AM | '01' |
| 02-22-1997 01:31:10AM | '04' |
| 10-24-1997 02:12:30PM | '04' |
| NULL | NULL |

The following expression returns the week of the year for each date in a column:

```
TO_CHAR( DATE_PROMISED, 'WW' )
```

| DATE_PROMISED | RETURN VALUE |
|---|---|
| 04-01-1997 12:00:10PM | '18' |
| 02-22-1997 01:31:10AM | '08' |
| 10-24-1997 02:12:30AM | '43' |
| NULL | NULL |

### Tip

You can combine TO_CHAR and TO_DATE to convert a numeric value for a month into the text value for a month using a function such as:

```
TO_CHAR( TO_DATE( numeric_month, 'MM' ), 'MONTH' )
```

# TO_CHAR (Numbers)

Converts numeric values to text strings. TO_CHAR also converts dates to strings.

TO_CHAR converts numeric values to text strings as follows:

- Converts double values to strings of up to 16 digits and provides accuracy up to 15 digits. If you pass a number with more than 15 digits, TO_CHAR rounds the number to the sixteenth digit.
- Returns decimal notation for numbers in the ranges (-1e16,-1e-16] and [1e-16, 1e16). TO_CHAR returns scientific notation for numbers outside these ranges.

**Note:** Data Integration converts the values 1e-16 and -1e16 to scientific notation, but returns the values 1e-16 and -1e-16 in decimal notation.

### Syntax

```
TO_CHAR( numeric_value )
```

| Argument | Required/ Optional | Description |
|---|---|---|
| *numeric_value* | Required | Numeric datatype. The numeric value you want to convert to a string. You can enter any valid expression. |

### Return Value

String.

NULL if a value passed to the function is NULL.

## Example

The following expression converts the values in the SALES column to text:

```
TO_CHAR( SALES )
```

| SALES | RETURN VALUE |
|---|---|
| 1010.99 | '1010.99' |
| -15.62567 | '-15.62567' |
| 10842764968208837340 | '1.084276496820884e+019' *(rounded to 16th digit)* |
| 1.234567890123456789e-10 | '0.000000000123456789012345 7' *(greater than 1e-16 but less than 1e16)* |
| 1.23456789012345e17 | '1.23456789012345e17' *(greater than 1e16)* |
| 0 | '0' |
| 33.15 | '33.15' |
| NULL | NULL |

# TO_DATE

Converts a character string to a date datatype in the same format as the character string. You use the TO_DATE format strings to specify the format of the source strings.

The target column must be date/time for TO_DATE expressions.

If you are converting two-digit years with TO_DATE, use either the RR or YY format string. Do not use the YYYY format string.

## Syntax

```
TO_DATE( string [, format] )
```

| Argument | Required/ Optional | Description |
|---|---|---|
| *string* | Required | Must be a string datatype. Passes the values that you want to convert to dates. You can enter any valid expression. |
| *format* | Optional | Enter a valid TO_DATE format string. The format string must match the parts of the *string* argument. For example, if you pass the string 'Mar 15 1998 12:43:10AM', you must use the format string 'MON DD YYYY HH12:MI:SSAM'. If you omit the format string, the string value must be in the default date of MM/DD/YYYY HH24:MI:SS.<br>For more information about TO_DATE format strings, see "TO_DATE and IS_DATE format strings" on page 37. |

## Return Value

Date.

TO_DATE always returns a date and time. If you pass a string that does not have a time value, the date returned always includes the time 00:00:00. You can map the results of this function to any target column with a date datatype.

NULL if you pass a null value to this function.

**Warning:** The format of the TO_DATE string must match the format string including any date separators. If it does not, Data Integration might return inaccurate values or skip the row.

## Example

The following expression returns date values for the strings in the DATE_PROMISED column. TO_DATE always returns a date and time. If you pass a string that does not have a time value, the date returned always includes the time 00:00:00. If you run a session in the twentieth century, the century will be 19. The current year on the machine running Data Integration is 1998:

```
TO_DATE( DATE_PROMISED, 'MM/DD/YY' )
```

| DATE_PROMISED | RETURN VALUE |
|---|---|
| '01/22/98' | Jan 22 1998 00:00:00 |
| '05/03/98' | May 3 1998 00:00:00 |
| '11/10/98' | Nov 10 1998 00:00:00 |
| '10/19/98' | Oct 19 1998 00:00:00 |
| NULL | NULL |

The following expression returns date and time values for the strings in the DATE_PROMISED column. If you pass a string that does not have a time value, Data Integration writes the row into the error rows file. If you run a session in the twentieth century, the century will be 19. The current year on the machine running Data Integration is 1998:

```
TO_DATE( DATE_PROMISED, 'MON DD YYYY HH12:MI:SSAM' )
```

| DATE_PROMISED | RETURN VALUE |
|---|---|
| 'Jan 22 1998 02:14:56PM' | Jan 22 1998 02:14:56PM |
| 'Mar 15 1998 11:11:11AM' | Mar 15 1998 11:11:11AM |
| 'Jun 18 1998 10:10:10PM' | Jun 18 1998 10:10:10PM |
| 'October 19 1998' | None. Data Integration writes the row into the error rows file. |
| NULL | NULL |

The following expression converts strings in the SHIP_DATE_MJD_STRING column to date values in the default date format:

```
TO_DATE (SHIP_DATE_MJD_STR, 'J')
```

| SHIP_DATE_MJD_STR | RETURN_VALUE |
|---|---|
| '2451544' | Dec 31 1999 00:00:00 |
| '2415021' | Jan 1 1900 00:00:00 |

Because the J format string does not include the time portion of a date, the return values have the time set to 00:00:00.

The following expression converts a string to a four-digit year format. The current year is 1998:

```
TO_DATE( DATE_STR, 'MM/DD/RR')
```

| DATE_STR | RETURN VALUE |
|---|---|
| '04/01/98' | 04/01/1998 00:00:00 |
| '08/17/05' | 08/17/2005 00:00:00 |

The following expression converts a string to a four-digit year format. The current year is 1998:

```
TO_DATE( DATE_STR, 'MM/DD/YY')
```

| DATE_STR | RETURN VALUE |
|---|---|
| '04/01/98' | 04/01/1998 00:00:00 |
| '08/17/05' | 08/17/1905 00:00:00 |

**Note:** For the second row, RR returns the year 2005 and YY returns the year 1905.

The following expression converts a string to a four-digit year format. The current year is 1998:

```
TO_DATE( DATE_STR, 'MM/DD/Y')
```

| DATE_STR | RETURN VALUE |
|---|---|
| '04/01/8' | 04/01/1998 00:00:00 |
| '08/17/5' | 08/17/1995 00:00:00 |

The following expression converts a string to a four-digit year format. The current year is 1998:

```
TO_DATE( DATE_STR, 'MM/DD/YYY')
```

| DATE_STR | RETURN VALUE |
|---|---|
| '04/01/998' | 04/01/1998 00:00:00 |
| '08/17/995' | 08/17/1995 00:00:00 |

The following expression converts strings that includes the seconds since midnight to date values:

```
TO_DATE( DATE_STR, 'MM/DD/YYYY SSSSS')
```

| DATE_STR | RETURN_VALUE |
|---|---|
| '12/31/1999 3783' | 12/31/1999 01:02:03 |
| '09/15/1996 86399' | 09/15/1996 23:59:59 |

If the target accepts different date formats, use TO_DATE and IS_DATE with the DECODE function to test for acceptable formats. For example:

```
DECODE( TRUE,
 --test first format
   IS_DATE( CLOSE_DATE,'MM/DD/YYYY HH24:MI:SS' ),
   --if true, convert to date
```

```
    TO_DATE( CLOSE_DATE,'MM/DD/YYYY HH24:MI:SS' ),
--test second format; if true, convert to date
  IS_DATE( CLOSE_DATE,'MM/DD/YYYY'), TO_DATE( CLOSE_DATE,'MM/DD/YYYY' ),
--test third format; if true, convert to date
  IS_DATE( CLOSE_DATE,'MON DD YYYY'), TO_DATE( CLOSE_DATE,'MON DD YYYY'),
--if none of the above
  ERROR( 'NOT A VALID DATE') )
```

You can combine TO_CHAR and TO_DATE to convert a numeric value for a month into the text value for a month using a function such as:

```
TO_CHAR( TO_DATE( numeric_month, 'MM' ), 'MONTH' )
```

# TO_DECIMAL

Converts a string or numeric value to a decimal value. TO_DECIMAL ignores leading spaces.

## Syntax

```
TO_DECIMAL( value [, scale] )
```

| Argument | Required/ Optional | Description |
|---|---|---|
| *value* | Required | Must be a string or numeric datatype. Passes the values you want to convert to decimals. You can enter any valid expression. |
| scale | Required or Optional | Must be an integer literal between 0 and 28, inclusive. Specifies the number of digits allowed after the decimal point. If you omit this argument, the function returns a value with the same scale as the input value.<br>You cannot omit this argument in advanced mode. |

## Return Value

Decimal of precision and scale between 0 and 28, inclusive.

0 if the value in the selected column is an empty string or a non-numeric character.

NULL if a value passed to the function is NULL.

## Example

This expression uses values from the column IN_TAX. The data type is decimal with precision of 10 and scale of 3:

```
TO_DECIMAL( IN_TAX, 3 )
```

| IN_TAX | RETURN VALUE |
|---|---|
| '15.6789' | 15.679 |
| '60.2' | 60.200 |
| '118.348' | 118.348 |
| NULL | NULL |

| IN_TAX | RETURN VALUE |
|---|---|
| 'A12.3Grove' | 0 |

# TO_FLOAT

Converts a string or numeric value to a double-precision floating point number (the Double datatype). TO_FLOAT ignores leading spaces.

## Syntax

```
TO_FLOAT( value )
```

| Argument | Required/ Optional | Description |
|---|---|---|
| *value* | Required | Must be a string or numeric datatype. Passes the values you want to convert to double values. You can enter any valid expression. |

## Return Value

Double value.

0 if the value in the column is blank or a non-numeric character.

NULL if a value passed to this function is NULL.

## Example

This expression uses values from the column IN_TAX:

```
TO_FLOAT( IN_TAX )
```

| IN_TAX | RETURN VALUE |
|---|---|
| '15.6789' | 15.6789 |
| '60.2' | 60.2 |
| '118.348' | 118.348 |
| NULL | NULL |
| 'A12.3Grove' | 0 |

# TO_INTEGER

Converts a string or numeric value to an integer. TO_INTEGER syntax contains an optional argument that you can choose to round the number to the nearest integer or truncate the decimal portion. TO_INTEGER ignores leading spaces.

## Syntax

```
TO_INTEGER( value [, flag] )
```

| Argument | Required/ Optional | Description |
|---|---|---|
| *value* | Required | String or numeric datatype. Passes the value you want to convert to an integer. You can enter any valid expression. |
| *flag* | Optional | Specifies whether to truncate or round the decimal portion. The flag must be an integer literal or the constants TRUE or FALSE:<br>- TO_INTEGER truncates the decimal portion when the flag is TRUE or a number other than 0.<br>- TO_INTEGER rounds the value to the nearest integer if the flag is FALSE or 0 or if you omit this argument. |

## Return Value

Integer.

NULL if the value passed to the function is NULL.

0 if the value passed to the function contains alphanumeric characters.

## Example

The following expressions use values from the column IN_TAX:

```
TO_INTEGER( IN_TAX, TRUE )
```

| IN_TAX | RETURN VALUE |
|---|---|
| '15.6789' | 15 |
| '60.2' | 60 |
| '118.348' | 118 |
| NULL | NULL |
| 'A12.3Grove' | 0 |
| '  123.87' | 123 |
| '-15.6789' | -15 |
| '-15.23' | -15 |

```
TO_INTEGER( IN_TAX, FALSE)
```

| IN_TAX | RETURN VALUE |
|---|---|
| '15.6789' | 16 |
| '60.2' | 60 |
| '118.348' | 118 |
| NULL | NULL |

| IN_TAX | RETURN VALUE |
|---|---|
| 'A12.3Grove' | 0 |
| '  123.87' | 124 |
| '-15.6789' | -16 |
| '-15.23' | -15 |

# TRUNC (Dates)

Truncates dates to a specific year, month, day, hour, or minute. You can also use TRUNC to truncate numbers.

You can truncate the following date parts:

- **Year.** If you truncate the year portion of the date, the function returns Jan 1 of the input year with the time set to 00:00:00. For example, the expression `TRUNC(6/30/1997 2:30:55, 'YY')` returns 1/1/1997 00:00:00, and `TRUNC(12/1/1997 3:10:15, 'YY')` returns 1/1/1997 00:00:00.
- **Month.** If you truncate the month portion of a date, the function returns the first day of the month with the time set to 00:00:00. For example, the expression `TRUNC(4/15/1997 12:15:00, 'MM')` returns 4/1/1997 00:00:00, and `TRUNC(4/30/1997 3:15:46, 'MM')` returns 4/1/1997 00:00:00.
- **Day.** If you truncate the day portion of a date, the function returns the date with the time set to 00:00:00. For example, the expression `TRUNC(6/13/1997 2:30:45, 'DD')` returns 6/13/1997 00:00:00, and `TRUNC(12/13/1997 22:30:45, 'DD')` returns 12/13/1997 00:00:00.
- **Hour.** If you truncate the hour portion of a date, the function returns the date with the minutes and seconds set to 0. For example, the expression `TRUNC(4/1/1997 11:29:35, 'HH')` returns 4/1/1997 11:00:00, and `TRUNC(4/1/1997 13:39:00, 'HH')` returns 4/1/1997 13:00:00.
- **Minute.** If you truncate the minute portion of a date, the function returns the date with the seconds set to 0. For example, the expression `TRUNC(5/22/1997 10:15:29, 'MI')` returns 5/22/1997 10:15:00, and `TRUNC(5/22/1997 10:18:30, 'MI')` returns 5/22/1997 10:18:00.

## Syntax

```
TRUNC( date [,format] )
```

| Argument | Required/ Optional | Description |
|---|---|---|
| *date* | Required | Date/Time datatype. The date values you want to truncate. You can enter any valid expression that evaluates to a date. |
| *format* | Optional | Enter a valid format string. The format string is not case sensitive. If you omit the format string, the function truncates the time portion of the date, setting it to 00:00:00. |

## Return Value

Date.

NULL if a value passed to the function is NULL.

## Example

The following expressions truncate the year portion of dates in the DATE_SHIPPED column:

```
TRUNC( DATE_SHIPPED, 'Y' )
TRUNC( DATE_SHIPPED, 'YY' )
TRUNC( DATE_SHIPPED, 'YYY' )
TRUNC( DATE_SHIPPED, 'YYYY' )
```

| DATE_SHIPPED | RETURN VALUE |
|---|---|
| Jan 15 1998 2:10:30AM | Jan 1 1998 12:00:00AM |
| Apr 19 1998 1:31:20PM | Jan 1 1998 12:00:00AM |
| Jun 20 1998 3:50:04AM | Jan 1 1998 12:00:00AM |
| Dec 20 1998 3:29:55PM | Jan 1 1998 12:00:00AM |
| NULL | NULL |

The following expressions truncate the month portion of each date in the DATE_SHIPPED column:

```
TRUNC( DATE_SHIPPED, 'MM' )
TRUNC( DATE_SHIPPED, 'MON' )
TRUNC( DATE_SHIPPED, 'MONTH' )
```

| DATE_SHIPPED | RETURN VALUE |
|---|---|
| Jan 15 1998 2:10:30AM | Jan 1 1998 12:00:00AM |
| Apr 19 1998 1:31:20PM | Apr 1 1998 12:00:00AM |
| Jun 20 1998 3:50:04AM | Jun 1 1998 12:00:00AM |
| Dec 20 1998 3:29:55PM | Dec 1 1998 12:00:00AM |
| NULL | NULL |

The following expressions truncate the day portion of each date in the DATE_SHIPPED column:

```
TRUNC( DATE_SHIPPED, 'D' )
TRUNC( DATE_SHIPPED, 'DD' )
TRUNC( DATE_SHIPPED, 'DDD' )
TRUNC( DATE_SHIPPED, 'DY' )
TRUNC( DATE_SHIPPED, 'DAY' )
```

| DATE_SHIPPED | RETURN VALUE |
|---|---|
| Jan 15 1998 2:10:30AM | Jan 15 1998 12:00:00AM |
| Apr 19 1998 1:31:20PM | Apr 19 1998 12:00:00AM |
| Jun 20 1998 3:50:04AM | Jun 20 1998 12:00:00AM |
| Dec 20 1998 3:29:55PM | Dec 20 1998 12:00:00AM |
| Dec 31 1998 11:59:59PM | Dec 31 1998 12:00:00AM |
| NULL | NULL |

The following expressions truncate the hour portion of each date in the DATE_SHIPPED column:

```
TRUNC( DATE_SHIPPED, 'HH' )
TRUNC( DATE_SHIPPED, 'HH12' )
TRUNC( DATE_SHIPPED, 'HH24' )
```

| DATE_SHIPPED | RETURN VALUE |
|---|---|
| Jan 15 1998 2:10:31AM | Jan 15 1998 2:00:00AM |
| Apr 19 1998 1:31:20PM | Apr 19 1998 1:00:00PM |
| Jun 20 1998 3:50:04AM | Jun 20 1998 3:00:00AM |
| Dec 20 1998 3:29:55PM | Dec 20 1998 3:00:00PM |
| Dec 31 1998 11:59:59PM | Dec 31 1998 11:00:00AM |
| NULL | NULL |

The following expression truncates the minute portion of each date in the DATE_SHIPPED column:

```
TRUNC( DATE_SHIPPED, 'MI' )
```

| DATE_SHIPPED | RETURN VALUE |
|---|---|
| Jan 15 1998 2:10:30AM | Jan 15 1998 2:10:00AM |
| Apr 19 1998 1:31:20PM | Apr 19 1998 1:31:00PM |
| Jun 20 1998 3:50:04AM | Jun 20 1998 3:50:00AM |
| Dec 20 1998 3:29:55PM | Dec 20 1998 3:29:00PM |
| Dec 31 1998 11:59:59PM | Dec 31 1998 11:59:00PM |
| NULL | NULL |

# TRUNC (Numbers)

Truncates numbers to a specific digit. You can also use TRUNC to truncate dates.

## Syntax

```
TRUNC( numeric_value [, precision] )
```

| Argument | Required/ Optional | Description |
|---|---|---|
| *numeric_value* | Required | Numeric datatype. Passes the values you want to truncate. You can enter any valid expression that evaluates to a Numeric datatype. |
| *precision* | Optional | Can be a positive or negative integer. You can enter any valid expression that evaluates to an integer. The integer specifies the number of digits to truncate. |

If *precision* is a positive integer, TRUNC returns *numeric_value* with the number of decimal places specified by *precision*. If *precision* is a negative integer, TRUNC changes the specified digits to the left of the decimal point to zeros. If you omit the *precision* argument, TRUNC truncates the decimal portion of *numeric_value* and returns an integer.

If you pass a decimal *precision* value, Data Integration rounds *numeric_value* to the nearest integer before evaluating the expression.

## Return Value

Numeric value.

NULL if one of the arguments is NULL.

## Example

The following expressions truncate the values in the PRICE column:

```
    TRUNC( PRICE, 3 )
```

| PRICE | RETURN VALUE |
|---|---|
| 12.9995 | 12.999 |
| -18.8652 | -18.865 |
| 56.9563 | 56.956 |
| 15.9928 | 15.992 |
| NULL | NULL |

```
    TRUNC( PRICE, -1 )
```

| PRICE | RETURN VALUE |
|---|---|
| 12.99 | 10.0 |
| -187.86 | -180.0 |
| 56.95 | 50.0 |
| 1235.99 | 1230.0 |
| NULL | NULL |

```
    TRUNC( PRICE )
```

| PRICE | RETURN VALUE |
|---|---|
| 12.99 | 12.0 |
| -18.99 | -18.0 |
| 56.95 | 56.0 |
| 15.99 | 15.0 |
| NULL | NULL |

# UPPER

Converts lowercase string characters to uppercase.

## Syntax

```
UPPER( string )
```

| Argument | Required/ Optional | Description |
|---|---|---|
| *string* | Required | String datatype. Passes the values you want to change to uppercase text. You can enter any valid expression. |

## Return Value

Uppercase string. If the data contains multibyte characters, the return value depends on the code page of the Secure Agent that runs the task.

NULL if a value passed to the function is NULL.

## Example

The following expression changes all names in the FIRST_NAME column to uppercase:

```
UPPER( FIRST_NAME )
```

| FIRST_NAME | RETURN VALUE |
|---|---|
| Ramona | RAMONA |
| NULL | NULL |
| THOMAS | THOMAS |
| PierRe | PIERRE |
| Bernice | BERNICE |

# VARIANCE

Returns the variance of a value you pass to it. VARIANCE is used to analyze statistical data.

You can nest only one other aggregate function within VARIANCE, and the nested function must return a numeric data type. You cannot nest aggregate functions in advanced mode.

Use only in mapping tasks.

## Syntax

```
VARIANCE( numeric_value [, filter_condition ] )
```

| Argument | Required/ Optional | Description |
|---|---|---|
| *numeric_value* | Required | Numeric data type. Passes the values for which you want to calculate a variance. You can enter any valid transformation expression. |
| *filter_condition* | Optional | Limits the rows in the search. The filter condition must be a numeric value or evaluate to TRUE, FALSE, or NULL. You can enter any valid transformation expression. |

## Formula

The function uses the following formula to calculate the variance:

$$\frac{\sum_{i=1}^{n} x_i^2 - \frac{1}{n}\left[\sum_{i=1}^{n} x_i\right]^2}{n-1}$$

Use the following guidelines for this formula:

- *xi* is one of the numeric values.
- *n* is the number of elements in the set of numeric values. If n is 1, the variance is 0.

## Return Value

Double value.

NULL if all values passed to the function are NULL or if no rows are selected (for example, the *filter_condition* evaluates to FALSE or NULL for all rows).

## Nulls

If a single value is NULL, VARIANCE ignores it. However, if all values passed to the function are NULL or if no rows are selected, VARIANCE returns NULL.

## Group By

VARIANCE groups values based on group by fields you define in the transformation, returning one result for each group.

If there is not a group by field, VARIANCE treats all rows as one group, returning one value.

## Example

The following expression calculates the variance of all rows in the TOTAL_SALES field:

```
VARIANCE( TOTAL_SALES )
```

| TOTAL_SALES |
|---|
| 2198.0 |
| 2256.0 |
| 3001.0 |

**TOTAL_SALES**

NULL

8953.0

**RETURN VALUE:** 10592444.6666667

CHAPTER 6

# System variables

System variables return system values that change, such as the current task and run ID.

## CurrentMappingName

Returns the name of the mapping that the task is based on as a string value. Use CurrentMappingName with any function that accepts string datatypes.

Data Integration resolves CurrentMappingName when you run a task. It does not resolve CurrentMappingName when you test run a mapping. If you want to run a mapping and return the current mapping name, use CurrentTaskName.

## CurrentRunId

Each task run has a unique ID. CurrentRunId returns the system generated run ID for the mapping task as a string value. Use CurrentRunId with any function that accepts string datatypes.

## CurrentTaskId

Returns the task ID as a string value. Use CurrentTaskId with any function that accepts string datatypes.

## CurrentTaskName

Returns the task name as a string value. Use CurrentTaskName with any function that accepts string datatypes. Also returns the mapping name in a mapping run.

# SESSSTARTTIME

Returns the date and time that the job started running in the time zone of the agent that runs the job.

SESSSTARTTIME is stored as a transformation Date/Time data type value. Use SESSSTARTTIME with any function that accepts transformation Date/Time data types. You can use SESSSTARTTIME only within the expression language.

# SYSDATE

Returns the agent system date and time for each row that passes through the transformation.

SYSDATE is stored as a transformation Date/Time data type variable.

To return a static date and time, use the SESSSTARTTIME variable.

CHAPTER 7

# Datatype reference

This chapter includes the following topics:



## Datatype reference overview

When you create a mapping or task, you create a set of instructions for Data Integration to read data from a source and write it to a target. Data Integration transforms data based on the data flow and the datatype assigned to each field.

Data Integration uses the following types of datatypes:

**Native datatypes**

Native datatypes are specific to sources, targets, and lookup objects in a mapping or task. In a mapping, native datatypes appear in the sources, targets, and lookup objects.

**Transformation datatypes**

Transformation datatypes are internal datatypes that appear in all transformations in a mapping. They are based on ANSI SQL-92 generic datatypes. Data Integration uses transformation datatypes to move data across platforms.

In tasks that move data without transforming it, such as replication tasks, Data Integration converts the native datatypes from the source to the comparable native datatypes in the target. In mappings and mapping tasks, Data Integration converts the native datatypes from the source to the comparable transformation datatypes before it starts transforming the data. When Data Integration writes to a target, it converts the transformation datatypes to the comparable native datatypes.

When you specify a multibyte character set, the datatypes allocate additional space in the database to store characters of up to three bytes.

# Rules and guidelines for datatypes

Use the following rules and guidelines for datatypes and conversion:

- The default datatype for all fields in flat files is Nstring(255).
  If you need to change a datatype, use the **Edit Metadata** option to edit the native type or precision for the field.
- The task may have unexpected results if you map incompatible datatypes between source and target fields or between the output of an expression and a target field.
  For example, if you map a datetime column of a MySQL database source to an integer column of a Salesforce target in a synchronization task, the task fails.
- Numeric data with a precision greater than 28 is truncated to a precision of 15 when written to a target. For example, 4567823451237864532451.12345678 might be written to the target as 4567823451237864000000.00000000.
- A synchronization task fails when you map source fields of the following datatype and database type to a Salesforce target:

| Datatype | Database type |
|---|---|
| tinyint | MySQL |
| tinyint | SQL Server |
| interval year to month | Oracle |
| interval day to second | Oracle |
| rowid | Oracle |
| urowid | Oracle |

- In most cases, the replication task creates the same precision and scale in the target as the source. In other cases, the replication task uses a different precision or scale:
  - In some cases, the replication task does not specify the precision or scale, and the database uses the defaults. For example, the source is MySQL and source datatype is Double(12). The replication task creates the Number datatype in an Oracle target and does not specify the precision.
  - If the source is not Oracle, the target is Oracle, and the source datatype is nchar, nvarchar, or nclob, the application multiples the source field precision by 2, up to a maximum of 64000, to obtain the target field precision.
  - If the source is MySQL, the target is Microsoft SQL Server 2000 or 2005, and the source datatype is date or time, the target field datatype is Timestamp(23, 3).
- If a replication task writes data from MySQL to a flat file and the source contains time data, Data Integration converts the time data to a date/time datatype, where the date is the current date and the time is the time specified in the source.
  You can use a string function in an expression to remove the date before loading the flat file.
- For an Oracle source or target, the precision of a Number field must be greater than or equal to the scale. Otherwise, the task fails.
- When a synchronization task writes 17-digit or 18-digit numeric data with no scale from Salesforce to an Oracle column with a Number datatype, the task may produce unexpected output in the target.

For example, the synchronization task writes the Salesforce value 67890123456789045 as 67890123456789048 in an Oracle target.

- When you use an ODBC connection for an Oracle database target, ensure that the maximum precision for an Oracle table column does not exceed the following values: char(1999), varchar(3999), nvarchar(3998), and nchar(3998).
- A task may load corrupt data into the target if the data comes from a source field of the Real datatype from Microsoft SQL Server.

# Transformation datatypes

The following table describes the transformation datatypes:

| Datatype | Size in bytes | Description |
|---|---|---|
| Bigint | 8 bytes | -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807<br>Precision of 19, scale of 0<br>Integer value. |
| Binary | Precision | 1 to 104,857,600 bytes<br>You cannot use binary data for flat file sources. |
| Date/Time | 16 bytes | Jan 1, 0001 A.D. to Dec 31, 9999 A.D.<br>Precision of 29, scale of 9<br>(precision to the nanosecond)<br>Combined date/time value. |
| Decimal | 8 bytes (if high precision is off or precision is greater than 28)<br>16 bytes (if precision <= 18 and high precision is on)<br>20 bytes (if precision > 18 and <= 28) | Decimal value with declared precision and scale. Scale must be less than or equal to precision.<br>Precision 1 to 28 digits, scale 0 to 28 |
| Double | 8 bytes | Double-precision floating-point numeric value.<br>You can edit the precision and scale. The scale must be less than or equal to the precision. |
| Integer | 4 bytes | -2,147,483,648 to 2,147,483,647<br>Precision of 10, scale of 0<br>Integer value. |
| Nstring | Unicode mode: (precision + 1) * 2<br>ASCII mode: precision + 1 | 1 to 104,857,600 characters<br>Fixed-length or varying-length string. |
| Ntext | Unicode mode: (precision + 1) * 2<br>ASCII mode: precision + 1 | 1 to 104,857,600 characters<br>Fixed-length or varying-length string. |

| Datatype | Size in bytes | Description |
|---|---|---|
| Real | 8 bytes | Precision of 7, scale of 0<br>Double-precision floating-point numeric value. |
| Small Integer | 4 bytes | -32,768 and 32,767<br>Precision of 5, scale of 0<br>Integer value. |
| String | Unicode mode: (precision + 1) * 2<br>ASCII mode: precision + 1 | 1 to 104,857,600 characters<br>Fixed-length or varying-length string. |
| Text | Unicode mode: (precision + 1) * 2<br>ASCII mode: precision + 1 | 1 to 104,857,600 characters<br>Fixed-length or varying-length string. |

# Integer datatypes

You can pass integer data from sources to targets and perform transformations on integer data. Data Integration supports Bigint, Integer, and Small Integer datatypes.

The transformation integer datatypes represent exact values.

## Integer values in calculations

When you use integer values in calculations, Data Integration sometimes converts integer values to floating-point numbers before it performs the calculation.

For example, to evaluate `MOD( 12.00, 5 )`, Data Integration converts the integer value "5" to a floating-point number before it performs the division operation. Data Integration converts integer values to double or decimal values depending on how you set the **Enable High Precision** advanced session property in the mapping task.

Data Integration converts integer values in the following arithmetic operations:

| Operation | High Precision Disabled | High Precision Enabled |
|---|---|---|
| Functions and calculations that cannot introduce decimal points.<br>For example, integer addition, subtraction, and multiplication, and functions such as CUME and SUM. | No conversion.<br>However, if the calculation produces a result that is out of range, Data Integration writes a row error. | Decimal |
| Non-scientific functions and calculations that can introduce decimal points.<br>For example, integer division, and functions such as AVG, MEDIAN, and PERCENTILE. | Double | Decimal |
| All scientific functions and the EXP, LN, LOG, POWER, and SQRT functions. | Double | Double |

The transformation Double datatype supports precision of up to 15 digits, while the Bigint datatype supports precision of up to 19 digits. Therefore, precision loss can occur in calculations that produce Bigint values with precision of more than 15 digits.

For example, an expression transformation contains the following calculation:

```
POWER( BIGINTVAL, EXPVAL )
```

Before it performs the calculation, Data Integration converts the inputs to the POWER function to double values. If the BIGINTVAL field contains the Bigint value 9223372036854775807, Data Integration converts this value to 9.22337203685478e+18, losing the last 4 digits of precision. If the EXPVAL field contains the value 1.0 and the result field is a Bigint, the calculation produces a row error because the result, 9223372036854780000, exceeds the maximum Bigint value.

When you use an Integer datatype in a calculation that can produce decimal values and you enable high precision, Data Integration converts the integer values to decimal values.

For transformations that support the Decimal datatype with precision up to 28 digits, precision loss does not occur in a calculation unless the result produces a value with precision greater than 28 digits in high precision mode. In this case, Data Integration stores the result as a double. If the field precision is less than or equal to 28 digits and the result produces a value greater than 28 digits in high precision mode, Data Integration rejects the row.

## Integer constants in expressions

Data Integration interprets constants in an expression as floating-point values, even if the calculation produces an integer result.

For example, in the expression `INTVALUE + 1000`, Data Integration converts the integer value "1000" to a double value if high precision is not enabled. It converts the value 1000 to a decimal value if high precision is enabled. To process the value 1000 as an integer value, create a variable field with an Integer datatype to hold the constant, and modify the expression to add the two fields.

## NaN values

NaN (Not a Number) is a value that is usually returned as the result of an operation on invalid input operands, especially in floating-point calculations. For example, when an operation attempts to divide zero by zero, it returns a NaN result.

Operating systems and programming languages may represent NaN differently. For example, the following list shows valid string representations of NaN:

- nan
- NaN
- NaN%
- NAN
- NaNQ
- NaNS
- qNaN
- sNaN
- 1.#QNAN
- 1.#SNAN

Data Integration converts QNAN values to 1.#QNAN on Win64EMT platforms. 1.#QNAN is a valid representation of NaN.

If an expression runs on an advanced cluster and evaluates to a NaN result, the return value for the expression is blank in the output.

### Convert string values to integer values

When Data Integration performs implicit conversion of a string value to an integer value, it truncates the data at the first non-numeric character.

For example, you link a string field that contains the value "9,000,000,000,000,000,000.777" to a Bigint field. Data Integration converts the string to the Bigint value 9,000,000,000,000,000,000.

### Write integer values to flat files

When writing integer values to a fixed-width flat file, the file writer does not verify that the data is within range.

For example, the file writer writes the result 3,000,000,000 to a target Integer column if the field width of the target column is at least 13. The file writer does not reject the row because the result is outside the valid range for Integer values.

## Binary datatype

If a mapping includes binary data, set the precision for the transformation binary datatype so that Data Integration can allocate enough memory to move the data from source to target.

You cannot use binary datatypes for flat file sources.

Before you can use functions with binary data, add the INFA_ENABLE_BINARY_FUNCTIONS custom property to the Data Integration Server service in Administrator. The following table shows how to configure this property:

| Service | Type | Sub-type | Name | Value | Sensitive |
|---|---|---|---|---|---|
| Data Integration Server | DTM | - | INFA_ENABLE_BINARY_FUNCTIONS | True | - |

## Date/Time data type

The Date/Time data type handles years from 1 A.D. to 9999 A.D. in the Gregorian calendar system. Years beyond 9999 A.D. cause an error. Mappings process Date/Time data types in complex file formats such as Parquet, Avro, and ORC using the local time zone of the Secure Agent machine.

The Date/Time data type supports dates with precision to the nanosecond. The data type has a precision of 29 and a scale of 9. Some native data types have a smaller precision. When you import a source that contains datetime values, the import process imports the correct precision from the source column. For example, the Microsoft SQL Server Datetime data type has a precision of 23 and a scale of 3. When you import a Microsoft SQL Server source that contains Datetime values, the Datetime columns in the mapping source have a precision of 23 and a scale of 3.

Data Integration reads datetime values from the source to the precision specified in the mapping source. When Data Integration transforms the datetime values, it supports precision up to 29 digits. For example, if you import a datetime value with precision to the millisecond, you can use the ADD_TO_DATE function in an Expression transformation to add nanoseconds to the date.

If you write a Date/Time value to a target column that supports a smaller precision, Data Integration truncates the value to the precision of the target column. If you write a Date/Time value to a target column that supports a larger precision, Data Integration inserts zeroes in the unsupported portion of the datetime value.

An advanced cluster processes Date/Time precision to the microsecond. If a Date/Time value contains nanoseconds, the advanced cluster truncates trailing digits.

# Decimal and double data types

You can pass decimal and double data from sources to targets and perform transformations on decimal and double data.

Data Integration supports the following data types:

**Decimal**

Precision 1 to 28 digits, scale 0 to 28. You cannot use decimal values with scale greater than precision or a negative precision. Transformations display any range that you assign to a decimal data type, but Data Integration supports precision only up to 28 digits.

When you enable high precision and the field precision is greater than 28 digits, Data Integration stores the result as a double.

**Double**

Double-precision floating-point numeric value.

You can edit the precision and scale. The scale must be less than or equal to the precision.

## Decimal and double values in calculations

Precision loss can occur with decimal and double data types in a calculation when the result produces a value with a precision greater than the maximum.

If you disable high precision, Data Integration converts decimal values to double. Precision loss occurs if the decimal value has a precision greater than 15 digits. For example, you have a mapping with decimal (20,0) that passes the number 40012030304957666903. If you disable high precision, Data Integration converts the decimal value to double and passes $4.00120303049577 \times 10^{19}$.

For transformations that support decimal data type of precision up to 28 digits, use the decimal data type and enable high precision to ensure precision of up to 28 digits.

Precision loss does not occur in a calculation unless the result produces a value with precision greater than the maximum allowed digits. In this case, Data Integration stores the result as a double.

Do not use the double data type for data that you use in an equality condition, such as a lookup or join condition.

The following table lists how Data Integration handles decimal values based on how the **Enable High Precision** advanced session property is set:

| Field Data type | Precision | High Precision Disabled | High Precision Enabled |
|---|---|---|---|
| Decimal | 0 to 15 | Decimal | Decimal |
| Decimal | 15 to 28 | Double | Decimal |
| Decimal | Over 28 | Double | Double |

When you enable high precision, Data Integration converts numeric constants in any expression function to decimal. If you do not enable high precision, Data Integration converts numeric constants to double.

You can ensure the maximum precision for numeric values greater than 28 or 38 digits depending on the transformation. Before you perform any calculations or transformations with the transformation functions, truncate or round any large numbers.

### Rounding for double values

Depending on the datatype of the downstream field, Data Integration rounds the double datatype value to the 15th digit.

The double datatype conforms to the IEEE 794 standard. Changes to database client library, different versions of a database or changes to a system run-time library affect the binary representation of mathematically equivalent values. Also, many system run-time libraries implement the round-to-even or the symmetric arithmetic method. The round-to-even method states that if a number falls midway between the next higher or lower number it round to the nearest value with an even least significant bit. For example, with the round-to-even method, 0.125 is rounded to 0.12. The symmetric arithmetic method rounds the number to next higher digit when the last digit is 5 or greater. For example, with the symmetric arithmetic method 0.125 is rounded to 0.13 and 0.124 is rounded to 0.12.

If the double datatype field in the Source transformation is connected to a decimal or string datatype field in the downstream transformation, Data Integration retains digits beyond the 15 significant digits without rounding. For all other downstream field datatypes, Data Integration rounds double datatype values to the 15th digit. For example, if a calculation on Windows returns the number 1234567890.1234567890, and the same calculation on UNIX returns 1234567890.1234569999, Data Integration converts this number to 1234567890.1234600000.

### Decimal and double values in advanced mode

In advanced mode, mappings use either high or low precision based on the types of hierarchical fields in the mapping. If the mapping has hierarchical fields that contain child fields with decimal data types, the mapping runs using low precision. Otherwise, the mapping runs using high precision.

When a mapping in advanced mode uses high precision, the advanced cluster handles decimal data types as decimals up to a precision of 38.

An advanced cluster and the Data Integration Server scale decimal values differently. The Data Integration Server allows the scale to differ among rows of decimal data, while an advanced cluster uses a fixed scale for each row. For example, when an advanced cluster processes the decimal 1.1234567 with scale 9, the output is 1.123456700. However, on the Data Integration Server, the output is 1.1234567.

When a mapping in advanced mode writes data to a flat file, decimal source values in exponential notation and with a scale of more than six are written without exponential notation to the target. For example, the decimal value 0E-20 in the source is written as 0.00000000000000000000 to the target.

## String datatypes

The transformation string datatypes include Nstring, Ntext, String and Text. Although the Nstring, Ntext, String, and Text datatypes support the same precision up to 104,857,600 characters, Data Integration uses String to move string data from source to target and Text to move text data from source to target.

Because some databases store text data differently than string data, Data Integration needs to distinguish between the two types of character data. If the source displays String, set the target column to String. Likewise, if the source displays Text, set the target column to Text, Long, or Long Varchar, depending on the source.

In general, the smaller string datatypes, such as Char and Varchar, display as String in the source, Lookup transformation, and SQL transformation, while the larger text datatypes, such as Text, Long, and Long Varchar, display as Text in the source.

Use Nstring, Ntext, String, and Text interchangeably within transformations. However, in the source, Lookup transformation, and SQL transformation, the target datatypes must match. The database drivers need to match the string datatypes with the transformation datatypes, so that the data passes accurately. For example, Nchar in a lookup table must match Nstring in the Lookup transformation.

# INDEX

# D

# J



# L



# M



# N

# O



# P



# R



# S

# T

# U



# V



# W



# Y