TECHNICAL REFERENCE

# Functions in Report Expressions

A complete reference to every function available in the Elda Reporting Expression Editor — grouped by what it does, with the syntax and an example.

bitberry

**Bitberry Dev**

Version 1.0 · September 2026

bitberry

# Functions in Report Expressions

This guide lists every function you can use in the Elda Reporting Expression Editor. Section 1 covers the syntax that applies to all of them; the remaining sections are the reference itself, grouped by the kind of work each function does.




PREPARED BY
Bitberry Dev

VERSION
1.0 · September 2026

APPLIES TO
Elda Reporting — the Report Designer expression editor, calculated fields, summary fields and parameter bindings

SECTION 1

# How report expressions work

Where expressions are used, and the syntax rules that apply to every function in this guide.

An **expression** is a small formula you type into the Report Designer's Expression Editor. The report evaluates it while the document is generated and uses the result — as the text of a label, the visibility of a band, the colour of a cell, the value of a calculated field, or the value passed to a query parameter.

Expressions are built from four ingredients: *references* to your data, *literal* values you type in, *operators*, and the *functions* catalogued in the rest of this document.

### Data fields

Square brackets refer to a field in the bound data source.

```
[UnitPrice]
```

### Parameters

A question mark refers to a report parameter.

```
?ProductCategory
```

### Text and dates

Text goes in single quotes; dates and times go between hashes.

```
'Bob'  #2022-01-01 00:00:00#
```

### Collections

A related collection can be filtered, then aggregated.

```
[Products][[Discontinued] == False]
```

## Reading the tables in this guide

Every function is listed with its signature. The words inside the brackets are placeholders describing what to supply — replace them with your own fields, parameters or literal values. Where a function appears more than once with different brackets, those are *overloads*: the same function accepting a different number of arguments. Where an example is shown, it uses the sample order and product data that ships with the Report Designer.

> **Not every function is available everywhere**
>
> The Expression Editor only offers the functions that make sense in the context you opened it from. The **summary functions** in section 9 appear only in the Summary Expression Editor; the **stored procedure functions** in section 11 appear only when you map report parameters to a stored procedure. If a function you expect is missing from the list, check where you opened the editor from.

SECTION 2

# Frequently used

The short list most report writers need day to day: relative dates, and shifting a date or time.

Most reporting expressions come down to two things: naming a point in time, and moving a date backwards or forwards from it. The tables below cover both. Everything here also appears in its full form in section 5.

## Relative dates

Each of these returns an actual date/time value, so they can be used on their own or as the starting point for one of the Add functions below.

| PARAMETER | RETURNS |
|---|---|
| `LocalDateTimeLastMonth()` | First day of previous month |
| `LocalDateTimeLastWeek()` | First day of previous week |
| `LocalDateTimeLastYear()` | First day of previous year |
| `LocalDateTimeNextMonth()` | First day of next month |
| `LocalDateTimeNextWeek()` | First day of next week |
| `LocalDateTimeNextYear()` | First day of next year |
| `LocalDateTimeNow()` | Current day and time |
| `LocalDateTimeThisMonth()` | First day of current month |
| `LocalDateTimeThisWeek()` | First day of current week |
| `LocalDateTimeThisYear()` | First day of current year |
| `LocalDateTimeToday()` | Current day |
| `LocalDateTimeTomorrow()` | Tomorrow's date |
| `LocalDateTimeTwoMonthsAway()` | First day of the month after next |
| `LocalDateTimeTwoWeeksAway()` | First day of the week after next |
| `LocalDateTimeTwoYearsAway()` | First day of the year after next |
| `LocalDateTimeYearBeforeToday()` | Date of one year ago from current day |
| `LocalDateTimeYesterday()` | Yesterday's date |
| `Today()` | Current day, midnight |

## Shifting a date or time

The first argument is the date or time to start from; the second is how far to move. A negative number counts backwards.

| FUNCTION | WHAT IT DOES |
| --- | --- |
| `AddDays(date, number of days)` | Adds a number of days to the specified date. A negative number counts backwards.<br>EXAMPLE `AddDays(Today(), -10) -- the date 10 days before today` |
| `AddHours(time, number of hours)` | Adds a number of hours to the specified time. A negative number counts backwards.<br>EXAMPLE `AddHours(LocalDateTimeNow(), -10) -- the time 10 hours before now` |
| `AddMonths(date, number of months)` | Adds a number of months to the specified date. A negative number counts backwards.<br>EXAMPLE `AddMonths(Today(), -10) -- the date 10 months before today` |

### Putting the two together

Combining them is how most date filters get written — for example `AddMonths(LocalDateTimeThisMonth(), -3)` gives the first day of the month three months back, which makes a clean starting boundary for a rolling quarter.

SECTION 3

# Aggregate functions

Calculate a single value from a collection of related records.

Aggregate functions run against a *collection* — typically a related detail table. You can filter the collection first by placing a condition in a second set of square brackets, then aggregate what remains.

| FUNCTION | WHAT IT DOES |
|---|---|
| `Avg(Value)` | Evaluates the average of the values in the collection.<br>EXAMPLE `[Products].Avg([UnitPrice])` |
| `Count()` | Returns the number of objects in a collection.<br>EXAMPLE `[Products].Count()` |
| `Exists()` | Determines whether the object exists in the collection.<br>EXAMPLE `[Categories][[CategoryID] == 7].Exists()` |
| `[Collection] [Condition].Join(Expression)` | Concatenates all Expression values in a collection that meet the optional Condition into a single string. Values are separated by a comma unless another separator is supplied.<br>EXAMPLE `[][[CategoryID] == [^.CategoryID]].Join([CompanyName])` |
| `[Collection] [Condition].Join(Expression, Separator)` | As above, but uses the supplied Separator between values instead of a comma.<br>EXAMPLE `[][[CategoryID] == [^.CategoryID]].Join([CompanyName], ';')` |
| `Max(Value)` | Returns the maximum expression value in a collection.<br>EXAMPLE `[Products].Max([UnitPrice])` |
| `Min(Value)` | Returns the minimum expression value in a collection.<br>EXAMPLE `[Products].Min([UnitPrice])` |
| `Single()` | Returns an object if it is the only element in a collection.<br>EXAMPLE `[Accounts].Single() is not null` |
| `Single(Expression)` | Returns the Expression if the collection contains only one object that meets the specified condition (optional).<br>EXAMPLE `[Collection].Single([Property1])` |
| `Sum(Value)` | Returns the sum of all expression values in the collection.<br>EXAMPLE `[Products].Sum([UnitsInStock])` |

SECTION 4

# Logical functions

Branch on a condition, and deal with missing values.

These are the workhorses of conditional formatting and conditional text. **`Iif`** chooses between values; the **`IsNull`** family keeps empty data from breaking a calculation or leaving a blank on the page.

| FUNCTION | WHAT IT DOES |
|---|---|
| **`Iif(Expression1, True_Value1, ..., ExpressionN, True_ValueN, False_Value)`** | Returns one of several specified values depending upon the values of logical expressions. The function accepts 2N+1 arguments: each odd argument is a logical test, each even argument is the value returned when the preceding test is true, and the final argument is returned when every test is false.<br>EXAMPLE `Iif(Name = 'Bob', 1, Name = 'Dan', 2, Name = 'Sam', 3, 4)` |
| **`IsNull(Value)`** | Returns True if the specified Value is NULL.<br>EXAMPLE `IsNull([OrderDate])` |
| **`IsNull(Value1, Value2)`** | Returns Value1 if it is not set to NULL; otherwise, Value2 is returned.<br>EXAMPLE `IsNull([ShipDate], [RequiredDate])` |
| **`IsNullOrEmpty(String)`** | Returns True if the specified String object is NULL or an empty string; otherwise, False is returned.<br>EXAMPLE `IsNullOrEmpty([ProductName])` |

SECTION 5

# Date and time functions

Test, build, shift and pull apart dates and times.

This is the largest group in the expression language. It is easiest to navigate by what you are trying to do: test whether a date falls in a period, take a date apart, shift it, measure the gap between two dates, or build a new one from separate values.

## Testing a date against a period

Each returns True or False, so these are well suited to a filter string or a conditional formatting rule.

| FUNCTION | WHAT IT DOES |
|---|---|
| `IsThisWeek(Date)` | Returns True if the specified date falls within the current week.<br>EXAMPLE `IsThisWeek([OrderDate])` |
| `IsThisMonth(Date)` | Returns True if the specified date falls within the current month.<br>EXAMPLE `IsThisMonth([OrderDate])` |
| `IsThisYear(Date)` | Returns True if the specified date falls within the current year.<br>EXAMPLE `IsThisYear([OrderDate])` |
| `IsLastMonth(Date)` | Returns True if the specified date falls within the previous month.<br>EXAMPLE `IsLastMonth([OrderDate])` |
| `IsLastYear(Date)` | Returns True if the specified date falls within the previous year.<br>EXAMPLE `IsLastYear([OrderDate])` |
| `IsNextMonth(Date)` | Returns True if the specified date falls within the next month.<br>EXAMPLE `IsNextMonth([OrderDate])` |
| `IsNextYear(Date)` | Returns True if the specified date falls within the next year.<br>EXAMPLE `IsNextYear([OrderDate])` |
| `IsYearToDate(Date)` | Returns True if the specified date falls within the period that starts from the first day of the current year and continues until the current date (including the current date).<br>EXAMPLE `IsYearToDate([OrderDate])` |
| `IsSameDay(Date1, Date2)` | Returns True if the specified date/time values fall within the same day.<br>EXAMPLE `IsSameDay([OrderDate], [ShippedDate])` |

| FUNCTION | WHAT IT DOES |
| --- | --- |
| `InDateRange(Date, FromDate, ToDate)` | Returns True if the date part of the first operand is greater than or equal to the date part of the second operand and less than or equal to the date part of the third operand. Otherwise, returns False. If the operands cannot be compared, returns null.<br>EXAMPLE `InDateRange([OrderDate], #2022-01-01 00:00:00#, #2022-12-31 23:59:59#)` |

## Testing the month

| FUNCTION | WHAT IT DOES |
| --- | --- |
| `IsJanuary(Date)` | Returns True if the specified date falls within January.<br>EXAMPLE `IsJanuary([OrderDate])` |
| `IsFebruary(Date)` | Returns True if the specified date falls within February.<br>EXAMPLE `IsFebruary([OrderDate])` |
| `IsMarch(Date)` | Returns True if the specified date falls within March.<br>EXAMPLE `IsMarch([OrderDate])` |
| `IsApril(Date)` | Returns True if the specified date falls within April.<br>EXAMPLE `IsApril([OrderDate])` |
| `IsMay(Date)` | Returns True if the specified date falls within May.<br>EXAMPLE `IsMay([OrderDate])` |
| `IsJune(Date)` | Returns True if the specified date falls within June.<br>EXAMPLE `IsJune([OrderDate])` |
| `IsJuly(Date)` | Returns True if the specified date falls within July.<br>EXAMPLE `IsJuly([OrderDate])` |
| `IsAugust(Date)` | Returns True if the specified date falls within August.<br>EXAMPLE `IsAugust([OrderDate])` |
| `IsSeptember(Date)` | Returns True if the specified date falls within September.<br>EXAMPLE `IsSeptember([OrderDate])` |
| `IsOctober(Date)` | Returns True if the specified date falls within October.<br>EXAMPLE `IsOctober([OrderDate])` |
| `IsNovember(Date)` | Returns True if the specified date falls within November.<br>EXAMPLE `IsNovember([OrderDate])` |

| FUNCTION | WHAT IT DOES |
| --- | --- |
| `IsDecember(Date)` | Returns True if the specified date falls within December.<br>EXAMPLE `IsDecember([OrderDate])` |

## Testing the time of day

These evaluate the time part of a value. Lunch time, work time and free time follow the working hours configured for the application.

| FUNCTION | WHAT IT DOES |
| --- | --- |
| `BeforeMidday(Time)` | Returns True if the specified time is before 12:00 PM.<br>EXAMPLE `BeforeMidday([StartTime])` |
| `AfterMidday(Time)` | Returns True if the specified time is after 12:00 PM.<br>EXAMPLE `AfterMidday([StartTime])` |
| `IsMorning(Time)` | Returns True if the specified time falls between 6:00 AM and 12:00 PM.<br>EXAMPLE `IsMorning([StartTime])` |
| `IsAfternoon(Time)` | Returns True if the specified time falls between 12:00 PM and 6:00 PM.<br>EXAMPLE `IsAfternoon([StartTime])` |
| `IsEvening(Time)` | Returns True if the specified time falls between 6:00 PM and 9:00 PM.<br>EXAMPLE `IsEvening([StartTime])` |
| `IsNight(Time)` | Returns True if the specified time falls between 9:00 PM and 9:00 AM.<br>EXAMPLE `IsNight([StartTime])` |
| `IsLunchTime(Time)` | Returns True if the specified time falls within the lunch time.<br>EXAMPLE `IsLunchTime([StartTime])` |
| `IsWorkTime(Time)` | Returns True if the specified time falls within work time.<br>EXAMPLE `IsWorkTime([StartTime])` |
| `IsFreeTime(Time)` | Returns True if the specified time falls within free time.<br>EXAMPLE `IsFreeTime([StartTime])` |
| `IsThisHour(Time)` | Returns True if the specified time falls within the hour.<br>EXAMPLE `IsThisHour([StartTime])` |
| `IsLastHour(Time)` | Returns True if the specified time falls within the last hour.<br>EXAMPLE `IsLastHour([StartTime])` |

| FUNCTION | WHAT IT DOES |
| --- | --- |
| `IsNextHour(Time)` | Returns True if the specified time falls within the next hour.<br>EXAMPLE `IsNextHour([StartTime])` |
| `IsSameHour(Time)` | Returns True if the specified time falls within the same hour.<br>EXAMPLE `IsSameHour([StartTime])` |
| `IsSameTime(Time)` | Returns True if the specified time falls within the same time of day (hour and minute).<br>EXAMPLE `IsSameTime([StartTime])` |

## Fixed points in time

Each returns an actual DateTime value rather than True or False, so they are useful as the boundaries of a date filter.

| FUNCTION | WHAT IT DOES |
| --- | --- |
| `LocalDateTimeNow()` | Returns the DateTime value that is the current moment in time.<br>EXAMPLE `AddDays(LocalDateTimeNow(), 5)` |
| `LocalDateTimeToday()` | Returns the DateTime value with the date part that is the start of the current day, and the time part of 00:00:00.<br>EXAMPLE `AddDays(LocalDateTimeToday(), 5)` |
| `LocalDateTimeYesterday()` | Returns the DateTime value with the date part that is the previous day, and the time part of 00:00:00.<br>EXAMPLE `AddDays(LocalDateTimeYesterday(), 5)` |
| `LocalDateTimeTomorrow()` | Returns the DateTime value with the date part that is the next day, and the time part of 00:00:00.<br>EXAMPLE `AddDays(LocalDateTimeTomorrow(), 5)` |
| `LocalDateTimeDayAfterTomorrow()` | Returns the DateTime value that has the date part that is two days after the current date, and the time part of 00:00:00.<br>EXAMPLE `AddDays(LocalDateTimeDayAfterTomorrow(), 5)` |
| `LocalDateTimeThisWeek()` | Returns the DateTime value with the date part that is the first day of the current week, and the time part of 00:00:00.<br>EXAMPLE `AddDays(LocalDateTimeThisWeek(), 5)` |
| `LocalDateTimeLastWeek()` | Returns the DateTime value that has the date part that is 7 days before the start of the current week, and the time part of 00:00:00.<br>EXAMPLE `AddDays(LocalDateTimeLastWeek(), 5)` |

| FUNCTION | WHAT IT DOES |
|---|---|
| `LocalDateTimeNextWeek()` | Returns the DateTime value that has the date part that is 7 days after the start of the current week, and the time part of 00:00:00.<br>EXAMPLE `AddDays(LocalDateTimeNextWeek(), 5)` |
| `LocalDateTimeTwoWeeksAway()` | Returns the DateTime value with the date part that is the first day of the week after the next week, and the time part of 00:00:00.<br>EXAMPLE `AddDays(LocalDateTimeTwoWeeksAway(), 5)` |
| `LocalDateTimeThisMonth()` | Returns the DateTime value with the date part that is the first day of the current month, and the time part of 00:00:00.<br>EXAMPLE `AddMonths(LocalDateTimeThisMonth(), 5)` |
| `LocalDateTimeLastMonth()` | Returns the DateTime value that has the date part that is one month before the current date, and the time part of 00:00:00.<br>EXAMPLE `AddMonths(LocalDateTimeLastMonth(), 5)` |
| `LocalDateTimeNextMonth()` | Returns the DateTime value that has the date part that is the first day of the next month, and the time part of 00:00:00.<br>EXAMPLE `AddMonths(LocalDateTimeNextMonth(), 5)` |
| `LocalDateTimeTwoMonthsAway()` | Returns the DateTime value with the date part that is the first day of the month after the next month, and the time part of 00:00:00.<br>EXAMPLE `AddMonths(LocalDateTimeTwoMonthsAway(), 5)` |
| `LocalDateTimeThisYear()` | Returns the DateTime value with the date part that is the first day of the current year, and the time part of 00:00:00.<br>EXAMPLE `AddYears(LocalDateTimeThisYear(), 5)` |
| `LocalDateTimeLastYear()` | Returns the DateTime value that has the date part that is the first day of the previous year, and the time part of 00:00:00.<br>EXAMPLE `AddYears(LocalDateTimeLastYear(), 5)` |
| `LocalDateTimeNextYear()` | Returns the DateTime value with the date part that corresponds to the first day of the next year, and the time part of 00:00:00.<br>EXAMPLE `AddYears(LocalDateTimeNextYear(), 5)` |
| `LocalDateTimeTwoYearsAway()` | Returns the DateTime value with the date part that is the first day of the year after the next year, and the time part of 00:00:00.<br>EXAMPLE `AddYears(LocalDateTimeTwoYearsAway(), 5)` |
| `LocalDateTimeYearBeforeToday()` | Returns the DateTime value with the date part that is the date one year ago, and the time part of 00:00:00.<br>EXAMPLE `AddYears(LocalDateTimeYearBeforeToday(), 5)` |

## Shifting a date or time

A negative count moves backwards.

| FUNCTION | WHAT IT DOES |
| --- | --- |
| `AddTicks(DateTime, TicksCount)` | Returns a DateTime value that is the specified number of ticks before or after a specified start date.<br>EXAMPLE `AddTicks([OrderDate], 500000)` |
| `AddMilliSeconds(Time, MilliSecondsCount)` | Returns a DateTime/TimeOnly value that is the specified number of milliseconds before or after a specified start date/time.<br>EXAMPLE `AddMilliSeconds([StartTime], 200)` |
| `AddSeconds(Time, SecondsCount)` | Returns a DateTime/TimeOnly value that is the specified number of seconds before or after a specified start date/time.<br>EXAMPLE `AddSeconds([StartTime], 120)` |
| `AddMinutes(Time, MinutesCount)` | Returns a DateTime/TimeOnly value that is the specified number of minutes before or after a specified start date/time.<br>EXAMPLE `AddMinutes([StartTime], 30)` |
| `AddHours(Time, HoursCount)` | Returns a DateTime/TimeOnly value that is the specified number of hours before or after a specified start date/time.<br>EXAMPLE `AddHours([StartTime], 5)` |
| `AddDays(Date, DaysCount)` | Returns a DateTime/DateOnly value that is the specified number of days before or after a specified start date.<br>EXAMPLE `AddDays([OrderDate], 10)` |
| `AddMonths(Date, MonthsCount)` | Returns a DateTime/DateOnly value that is the specified number of months before or after a specified start date.<br>EXAMPLE `AddMonths([OrderDate], 3)` |
| `AddYears(Date, YearsCount)` | Returns a DateTime/DateOnly value that is the specified number of years before or after a specified start date.<br>EXAMPLE `AddYears([OrderDate], 2)` |
| `AddTimeSpan(DateTime, TimeSpan)` | Returns a DateTime value that differs by a specified amount of time from a specified date.<br>EXAMPLE `AddTimeSpan([OrderDate], [Duration])` |

## Measuring the gap between two values

Each counts the number of boundaries crossed between the two values, not the elapsed duration.

| FUNCTION | WHAT IT DOES |
|---|---|
| `DateDiffTick(StartDate, EndDate)` | Returns the number of tick boundaries between the specified dates.<br>EXAMPLE `DateDiffTick([OrderDate], Now())` |
| `DateDiffMilliSecond(StartTime, EndTime)` | Returns the number of millisecond boundaries between the specified dates/times.<br>EXAMPLE `DateDiffMilliSecond([StartTime], Now())` |
| `DateDiffSecond(StartTime, EndTime)` | Returns the number of second boundaries between the specified dates/times.<br>EXAMPLE `DateDiffSecond([StartTime], Now())` |
| `DateDiffMinute(StartTime, EndTime)` | Returns the number of minute boundaries between the specified dates/times.<br>EXAMPLE `DateDiffMinute([StartTime], Now())` |
| `DateDiffHour(StartTime, EndTime)` | Returns the number of hour boundaries between the specified dates/times.<br>EXAMPLE `DateDiffHour([StartTime], Now())` |
| `DateDiffDay(StartDate, EndDate)` | Returns the number of day boundaries between the specified dates.<br>EXAMPLE `DateDiffDay([OrderDate], Now())` |
| `DateDiffMonth(StartDate, EndDate)` | Returns the number of month boundaries between the specified dates.<br>EXAMPLE `DateDiffMonth([OrderDate], Now())` |
| `DateDiffYear(StartDate, EndDate)` | Returns the number of year boundaries between the specified dates.<br>EXAMPLE `DateDiffYear([OrderDate], Now())` |

## Taking a date apart

| FUNCTION | WHAT IT DOES |
|---|---|
| `GetDate(Date)` | Returns the date part of the specified date.<br>EXAMPLE `GetDate([OrderDate])` |
| `GetYear(Date)` | Gets the year in the specified date.<br>EXAMPLE `GetYear([OrderDate])` |
| `GetMonth(Date)` | Gets the month in the specified date.<br>EXAMPLE `GetMonth([OrderDate])` |
| `GetDay(Date)` | Gets the day of the month in the specified date.<br>EXAMPLE `GetDay([OrderDate])` |

| FUNCTION | WHAT IT DOES |
|---|---|
| `GetDayOfYear(Date)` | Gets the day of the year in the specified date.<br>EXAMPLE `GetDayOfYear([OrderDate])` |
| `GetDayOfWeek(Date)` | Gets the day of the week in the specified date.<br>EXAMPLE `GetDayOfWeek([OrderDate])` |
| `GetTimeOfDay(DateTime)` | Gets the time part of the specified date.<br>EXAMPLE `GetTimeOfDay([OrderDate])` |
| `GetHour(Time)` | Returns the hours value in the specified date/time.<br>EXAMPLE `GetHour([StartTime])` |
| `GetMinute(Time)` | Returns the minutes value in the specified date/time.<br>EXAMPLE `GetMinute([StartTime])` |
| `GetSecond(Time)` | Returns the seconds value in the specified date/time.<br>EXAMPLE `GetSecond([StartTime])` |
| `GetMilliSecond(Time)` | Returns the milliseconds value in the specified date/time.<br>EXAMPLE `GetMilliSecond([StartTime])` |

## The current date and time

| FUNCTION | WHAT IT DOES |
|---|---|
| `Now()` | Returns the DateTime value that is the current date and time.<br>EXAMPLE `AddDays(Now(), 7)` |
| `Today()` | Returns a DateTime value that is the current date. The time part is set to 00:00:00.<br>EXAMPLE `AddDays(Today(), 3)` |
| `UtcNow()` | Returns a DateTime object that is the current date and time in Universal Coordinated Time (UTC).<br>EXAMPLE `AddDays(UtcNow(), 7)` |

## Building a date or time from separate values

Useful when your data holds the year, month and day in separate columns.

| FUNCTION | WHAT IT DOES |
|---|---|
| `DateTimeFromParts(Year, Month, Day)` | Returns a date value constructed from the specified Year, Month and Day.<br>EXAMPLE `DateTimeFromParts([Year], [Month], [Day])` |

| FUNCTION | WHAT IT DOES |
| --- | --- |
| `DateTimeFromParts(Year, Month, Day, Hour)` | Returns a date value constructed from the specified Year, Month, Day and Hour.<br>EXAMPLE `DateTimeFromParts([Year], [Month], [Day], [Hour])` |
| `DateTimeFromParts(Year, Month, Day, Hour, Minute)` | Returns a date value constructed from the specified Year, Month, Day, Hour and Minute.<br>EXAMPLE `DateTimeFromParts([Year], [Month], [Day], [Hour], [Minute])` |
| `DateTimeFromParts(Year, Month, Day, Hour, Minute, Second)` | Returns a date value constructed from the specified Year, Month, Day, Hour, Minute and Second.<br>EXAMPLE `DateTimeFromParts([Year], [Month], [Day], [Hour], [Minute], [Second])` |
| `DateOnlyFromParts(Year, Month, Day)` | Returns a DateOnly value constructed from the specified Year, Month and Day.<br>EXAMPLE `DateOnlyFromParts([Year], [Month], [Day])` |
| `TimeOnlyFromParts(Hour, Minute)` | Returns a TimeOnly value constructed from the specified hour and minute.<br>EXAMPLE `TimeOnlyFromParts([Hour], [Minute])` |
| `TimeOnlyFromParts(Hour, Minute, Second)` | Returns a TimeOnly value constructed from the specified hour, minute and seconds.<br>EXAMPLE `TimeOnlyFromParts([Hour], [Minute], [Second])` |
| `TimeOnlyFromParts(Hour, Minute, Second, Millisecond)` | Returns a TimeOnly value constructed from the specified hour, minute, seconds and milliseconds.<br>EXAMPLE `TimeOnlyFromParts([Hour], [Minute], [Second], [Millisecond])` |

SECTION 6

# Math functions

Rounding, arithmetic, powers, logarithms and trigonometry.

## Rounding and basic arithmetic

| FUNCTION | WHAT IT DOES |
|---|---|
| `Abs(Value)` | Returns the given numeric expression's absolute, positive value. |
| `Sign(Value)` | Returns the positive (+1), zero (0), or negative (-1) sign of the given expression. |
| `Round(Value)` | Rounds the given value to the nearest integer. |
| `Round(Value, Precision)` | Rounds the given value to the nearest integer, or to a specified number of decimal places. |
| `Ceiling(Value)` | Returns the smallest integer that is greater than or equal to the numeric expression. |
| `Floor(Value)` | Returns the largest integer less than or equal to the numeric expression. |
| `Max(Value1, Value2)` | Returns the maximum value from the specified values. |
| `Min(Value1, Value2)` | Returns the minimum value from the specified values. |
| `Power(Value, Power)` | Returns a specified number raised to a specified power. |
| `Sqr(Value)` | Returns the square root of a given number. |
| `BigMul(Value1, Value2)` | Returns an Int64 containing the full product of two specified 32-bit numbers. |
| `Rnd()` | Returns a random number that is less than 1, but greater than or equal to zero. |

## Exponents and logarithms

| FUNCTION | WHAT IT DOES |
|---|---|
| `Exp(Value)` | Returns the float expression's exponential value. |
| `Log(Value)` | Returns a specified number's natural logarithm. |
| `Log(Value, Base)` | Returns the logarithm of a specified number in a specified Base. |
| `Log10(Value)` | Returns a specified number's base 10 logarithm. |

## Trigonometry

All angles are expressed in radians.

| FUNCTION | WHAT IT DOES |
|---|---|
| `Sin(Value)` | Returns the sine of the angle defined in radians. |
| `Cos(Value)` | Returns the angle's cosine, in radians. |
| `Tan(Value)` | Returns the tangent of the angle defined in radians. |
| `Asin(Value)` | Returns a number's arcsine (the angle in radians, whose sine is the given float expression). |
| `Acos(Value)` | Returns a number's arccosine (the angle in radians, whose cosine is the given float expression). |
| `Atn(Value)` | Returns a number's arctangent (the angle in radians, whose tangent is the given float expression). |
| `Atn2(Value1, Value2)` | Returns the angle whose tangent is the quotient of two specified numbers in radians. |
| `Sinh(Value)` | Returns the hyperbolic sine of the angle defined in radians. |
| `Cosh(Value)` | Returns the angle's hyperbolic cosine, in radians. |
| `Tanh(Value)` | Returns the hyperbolic tangent of the angle defined in radians. |

## Type conversion

Use these when a value arrives as text, or when you need to control how a number is rounded or divided.

| FUNCTION | WHAT IT DOES |
|---|---|
| `ToInt(Value)` | Converts Value to an equivalent 32-bit signed integer. |
| `ToLong(Value)` | Converts Value to an equivalent 64-bit signed integer. |
| `ToDecimal(Value)` | Converts Value to an equivalent decimal number. |
| `ToDouble(Value)` | Converts Value to an equivalent 64-bit double-precision floating-point number. |
| `ToFloat(Value)` | Converts Value to an equivalent 32-bit single-precision floating-point number. |
| `ToStr(Value)` | Returns a string representation of an object. |

SECTION 7

# String functions

Combine, trim, search and reshape text.

## Changing case and cleaning up

| FUNCTION | WHAT IT DOES |
|---|---|
| `Upper(String)` | Returns String in uppercase. |
| `Lower(String)` | Returns String in lowercase. |
| `Trim(String)` | Removes all leading and trailing SPACE characters from String. |
| `Reverse(String)` | Reverses the order of elements within String. |

## Measuring and searching

Character positions are zero-based: the first character of a string is at position 0.

| FUNCTION | WHAT IT DOES |
|---|---|
| `Len(Value)` | Returns an integer containing either the number of characters in a string or the nominal number of bytes required to store a variable. |
| `Contains(String1, SubString1)` | Returns True if SubString1 occurs within String1; otherwise, False is returned. |
| `StartsWith(String1, SubString1)` | Returns True if the beginning of String1 matches SubString1; otherwise, False. |
| `EndsWith(String1, SubString1)` | Returns True if the end of String1 matches SubString1; otherwise, False is returned. |
| `CharIndex(String1, String2)` | Returns the starting position of String1 within String2, beginning from the zero character position to the end of a string. |
| `CharIndex(String1, String2, StartLocation)` | Returns the starting position of String1 within String2, beginning from the StartLocation character position to the end of a string. |

## Building and editing text

| FUNCTION | WHAT IT DOES |
|---|---|
| `Concat(String1, ... , StringN)` | Returns a string value containing the concatenation of the current string with any additional strings. |
| `Substring(String, StartPosition)` | Retrieves a substring from String. The substring starts at StartPosition. |
| `Substring(String, StartPosition, Length)` | Retrieves a substring from String. The substring starts at StartPosition and has a specified Length. |

| FUNCTION | WHAT IT DOES |
| --- | --- |
| `Insert(String1, StartPosition, String2)` | Inserts String2 into String1 at the position specified by StartPosition. |
| `Replace(String1, SubString2, String3)` | Returns a copy of String1, in which SubString2 has been replaced with String3. |
| `Remove(String, StartPosition)` | Deletes all the characters from this instance, beginning at a specified position. |
| `Remove(String, StartPosition, Length)` | Deletes a specified number of characters from this instance, beginning at a specified position. |
| `PadLeft(String, Length)` | Left-aligns the defined string's characters, padding its left side with white space characters up to a specified total length. |
| `PadLeft(String, Length, Char)` | Left-aligns the defined string's characters, padding its left side with the specified Char up to a specified total length. |
| `PadRight(String, Length)` | Right-aligns the defined string's characters, padding its left side with empty space characters up to a specified total length. |
| `PadRight(String, Length, Char)` | Right-aligns the defined string's characters, padding its left side with the specified Char up to a specified total length. |

## Characters and character codes

| FUNCTION | WHAT IT DOES |
| --- | --- |
| `Ascii(String)` | Returns the ASCII code value of the leftmost character in a character expression. |
| `Char(Number)` | Converts an integer ASCII code to a character. |

SECTION 8

# Report-specific functions

Colours, parameter display text, and reading neighbouring rows.

These functions exist only inside a report. They give an expression access to things that are not in the data source — the colour system, a parameter's display text, and the rows either side of the current one.

| FUNCTION | WHAT IT DOES |
|---|---|
| `Rgb(Red, Green, Blue)` | Returns a string defining a color using the Red, Green, and Blue color channel values. |
| `Argb(Alpha, Red, Green, Blue)` | Returns a string defining a color using the Alpha, Red, Green, and Blue color channel values. |
| `GetDisplayText(?parameterName)` | Returns a Display Text for a parameter's lookup value. For non-lookup parameters, this function returns a value converted to string. |
| `CurrentRowIndexInGroup()` | Returns the current row's index within the group. |
| `GroupIndex(level)` | Locates the parent group row at the specified nesting level and returns that row's index. |
| `NextRowColumnValue(columnName)` | Obtains the next row and returns the value from the specified column. |
| `PrevRowColumnValue(columnName)` | Obtains the previous row and returns the value from the specified column. |

### Neighbouring rows

`NextRowColumnValue` and `PrevRowColumnValue` are what you use to show a change between rows — a difference against the previous line, or a marker where a value is about to change. They read the report's own row sequence, so the result depends on the current sort order.

SECTION 9

# Summary functions

Totals, counts and statistics across a group, a page or the whole report.

These functions are offered by the **Summary Expression Editor**, which you open from a summary field rather than from an ordinary control. Each one is calculated across a *summary region* — a group, a page, or the report — which you choose separately from the expression itself. Functions beginning `sumD` count each distinct value once.

## Counting

| FUNCTION | WHAT IT DOES |
|---|---|
| `sumCount(Expression)` | Counts the number of values within the specified summary region (group, page, or report). In a simple scenario, you may not pass a parameter. |
| `sumDCount(Expression)` | Counts the number of distinct values within the specified summary region (group, page, or report). In a simple scenario, you may not pass a parameter. |
| `sumRecordNumber(Expression)` | Returns the current record number in the specified summary region (group, page, or report). |

## Totals and running totals

| FUNCTION | WHAT IT DOES |
|---|---|
| `sumSum(Expression)` | Calculates the total of all values within the specified summary region (group, page, or report). |
| `sumDSum(Expression)` | Calculates the total of all distinct values within the specified summary region (group, page, or report). |
| `sumRunningSum(Expression)` | Calculates the sum of all previous values displayed before the current data row with the current data row value. |
| `sumCarryoverSum(Expression)` | Calculates the carried forward and brought forward totals. |
| `sumPercentage(Expression)` | Calculates the percent ratio of the current data row's value to the total of all the values within the specified summary region (group, page, or report). |

## Averages and statistics

| FUNCTION | WHAT IT DOES |
|---|---|
| `sumAvg(Expression)` | Calculates the average of all values within the specified summary region (group, page, or report). |
| `sumDAvg(Expression)` | Calculates the average of all distinct values within the specified summary region (group, page, or report). |
| `sumWAvg(Expression, Expression)` | Calculates the weighted average of all values within the specified summary region (group, page, or report). |

| FUNCTION | WHAT IT DOES |
| --- | --- |
| `sumMedian(Expression)` | Finds the middle number within a sequence. If the total number of elements is odd, this function returns the value of the middle number in a sequence. If the total number of elements is even, this function returns the arithmetical mean of the two middle numbers. |
| `sumMax(Expression)` | Calculates the maximum of all values within the specified summary region (group, page, or report). |
| `sumMin(Expression)` | Calculates the minimum of all values within the specified summary region (group, page, or report). |
| `sumStdDev(Expression)` | Calculates the standard deviation of all values within the specified summary region (group, page, or report). |
| `sumStdDevP(Expression)` | Calculates the standard population deviation of all values within the specified summary region (group, page, or report). |
| `sumDStdDev(Expression)` | Calculates the standard deviation of all distinct values within the specified summary region (group, page, or report). |
| `sumDStdDevP(Expression)` | Calculates the standard population deviation of all distinct values within the specified summary region (group, page, or report). |
| `sumVar(Expression)` | Calculates the amount of variance for all values within the specified summary region (group, page, or report). |
| `sumVarP(Expression)` | Calculates the population variance of all values within the specified summary region (group, page, or report). |
| `sumDVar(Expression)` | Calculates the amount of variance for all distinct values within the specified summary region (group, page, or report). |
| `sumDVarP(Expression)` | Calculates the population variance of all distinct values within the specified summary region (group, page, or report). |

SECTION 10

# Expression bindings and calculated fields

The extra functions available when you build a binding or a calculated field.

The following functions are used to construct expression bindings and calculated fields.

| FUNCTION | WHAT IT DOES |
| --- | --- |
| `NewLine()` | Returns the newline string defined for the current environment. |
| `FormatString(Format, Value1, ... , ValueN)` | Returns the specified string with formatted field values. |
| `Rgb(Red, Green, Blue)` | Returns a string defining a color using the Red, Green, and Blue color channel values. |
| `Join()` | Concatenates the multi-value report parameter's values into a string. This function is useful when you bind a multi-value parameter to a label to display the parameter's values in a report. |

SECTION 11

# Stored procedure functions

Passing multi-value parameters into a stored procedure.

The following functions are used to bind a report to a stored procedure. Both deal with the same problem: a report parameter that holds several values, and a query parameter that expects one thing.

| FUNCTION | WHAT IT DOES |
| --- | --- |
| `Join(parameter)` | Concatenates the multi-value report parameter's values into a string. This function can be used when mapping multi-value report parameters to query parameters generated from a stored procedure's parameters. |
| `Join(parameter, separator)` | As above, using the supplied separator between values. |
| `CreateTable(Column1, ..., ColumnN)` | Creates a table from several multi-value parameters' values. This function can be used when mapping multi-value report parameters to the query parameter that is generated from a stored procedure's User Defined Table Type parameter. |