Amtiri

AMTIRI SCRIPT

Types and conversions

Generated on November 25, 2026

A value in a script is a number, text, a boolean, a list, a record, a date and time, or `null`. No type is declared in a script: each value carries its own kind, and each operator and function says what it accepts. This chapter lists the kinds, what arithmetic gives, how text is joined, what can be compared, and how to convert. The examples use the sample variables.

The runtime types

Kind	type_of gives	Comes from
Whole number	`Integer`	Whole-number literals, to_int, length, the parts of a date, the colour functions, a variable holding a whole number
Whole number	`Long`	round with one argument
Decimal	`Double`	Decimal literals, every sum, difference, product, quotient and power, most math functions, to_double
Decimal	`Float`	Some Modbus decoders, such as mbFloat
Text	`String`	Text literals, format, to_string, the string functions
Boolean	`Boolean`	`true`, `false`, comparisons, the logic functions
List	`ArrayList`, for a list a script builds; a list from a module may name another class	List literals, the collection functions, a Modbus master's array collector, a logger's history
Record	`Rec`	Record literals, the control functions
Date and time	`LocalDateTime`	now, date and the other date functions
No value	`null`	`null`, and the cases listed under Null

- **Whole numbers and decimals mix freely.** They add, compare and pick by value whatever their kind, so the table matters only where a function needs one kind, where case matches a value, or where a result is shown as text: `3` and `3.0` print differently.
- **A colour is a whole number.** See Colours.
- **A dashboard's date field writes a date without a time of day**, which `type_of` names `LocalDate`. See Dates and time zones.

```
type_of(COUNT) # "Integer"
type_of(LEVEL) # "Double"
type_of(COUNT + 1) # "Double"
type_of(round(LEVEL)) # "Long"
type_of(TAG) # "String"
type_of(PUMP_ON) # "Boolean"
type_of([1, 2]) # "ArrayList"
type_of({low: 10}) # "Rec"
type_of(START) # "LocalDateTime"
type_of(null) # null
```

Arithmetic

- **Every `+`, `-`, `*`, `/` and `^` gives a decimal**, even between whole numbers: `COUNT + 1` is `4.0`, and `7 / 2` is `3.5`.
- **Negation keeps the kind.** `-COUNT` is the whole number `-3`, and `-LEVEL` the decimal `-42.5`.
- **Division by zero is not an error.** It gives `Infinity`, `-Infinity`, or `NaN` for `0 / 0`, and so do some math functions, such as `sqrt(-1)`. Guard a divisor that can reach zero with ivl, and a square root with inn.
- **Decimals are binary fractions.** `0.1 + 0.2` is not exactly `0.3`. Round before comparing a computed decimal with a fixed one.
- **To get a whole number back**, to_int cuts the decimals off, towards zero, and round rounds to the nearest, halves upwards.

```
COUNT + 1 # 4.0
7 / 2 # 3.5
COUNT * 2 # 6.0
2 ^ 10 # 1024.0
-COUNT # -3
-LEVEL # -42.5
type_of(-5) # "Integer"
1 / 0 # Infinity
0 / 0 # NaN
ivl(LEVEL / 0, 0.0) # 0.0
0.1 + 0.2 # 0.30000000000000004
0.1 + 0.2 == 0.3 # false
round(0.1 + 0.2, 0.01) == 0.3 # true
to_int(LEVEL) # 42
to_int(-2.7) # -2
round(LEVEL) # 43
round(-2.5) # -2
round(LEVEL, 10) # 40.0
mod(-7, 3) # -1.0
```

Joining text

`+` joins text when its **left** operand is text: the right value is written out as text and appended. `*` and `/` do the same when their left operand is text, so `"ab" * 3` is `"ab3"`.

- **The left operand decides.** `1 + "a"` is an error, because a number on the left makes `+` an addition.

- **It runs left to right.** `"Count: " + COUNT + 1` appends `3`, then `1`. Put arithmetic in parentheses, and turn a whole result back into a whole number, or it shows as a decimal.
- **A `null` on the right is an error.** Guard a value that may be missing with nvl.
- **Something else on the left gives `null`.** When the left operand of `+`, `*` or `/` is neither a number nor text (`null`, a boolean, a date, a list or a record), the result is `null`. `-` gives `null` for any left operand that is not a number, while `^` fails.
- **For control over the text**, use format, with Java's format specifiers: `%s` for any value, `%d` for a whole number, `%.1f` for one decimal. `%d` refuses a decimal. The decimal separator of `%f` follows the instance's language, `application.locale`, as it was when the formula was compiled.

```
"a" + 1 # "a1"
TAG + ": " + LEVEL # "P-101: 42.5"
"Pump " + PUMP_ON # "Pump true"
"Count: " + COUNT + 1 # "Count: 31"
"Count: " + (COUNT + 1) # "Count: 4.0"
"Count: " + to_int(COUNT + 1) # "Count: 4"
"ab" * 3 # "ab3"
1 + "a" # error
1 + 2 + "a" # error
"a" + null # error
"a" + nvl(null, "-") # "a-"
null + 1 # null
PUMP_ON + 1 # null
START + 1 # null
"%d pumps".format(COUNT) # "3 pumps"
"%d pumps".format(COUNT * 2) # error
"%s m".format(LEVEL) # "42.5 m"
```

Comparisons across types

Comparing	Gives
Two numbers, of any kind	Compared by value: `1 == 1.0`
`Infinity` or `NaN` with a number	The rules of floating point: `Infinity` is above every number, and `NaN` is equal to nothing, itself included
Two texts	Compared character by character; case matters
Two booleans	`false` sorts before `true`
Two dates and times	Compared in time order
`null` with `null`	Equal
`null` with a value	Not equal; `null` sorts before any value
A number with text, or any other two different kinds	An error
Two lists, or two records	An error, even with `==`

Two functions choose by comparing too:

- max, min, clamp, arrayMax and arrayMin compare numbers by value, and give back the chosen argument as it was: `clamp(LEVEL, 0, 40)` is the whole number `40`.

- case matches only an exactly equal value, of the same kind: a whole number never matches a decimal. Since arithmetic gives decimals, `case(COUNT + 1, 4, …)` finds no match.

```
1 == 1.0 # true
COUNT == 3.0 # true
TAG == "p-101" # false
"abc" < "abd" # true
false < true # true
START < START.addHours(1) # true
null == null # true
COUNT == null # false
null < 1 # true
1 / 0 > 5 # true
0 / 0 == 0 / 0 # false
0 / 0 <> 1 # true
"1" == 1 # error
1 == "1" # error
[1] == [1] # error
max(1, 2.5) # 2.5
clamp(-1, 0, 10) # 0
clamp(LEVEL, 0, 40) # 40
case(COUNT, 3, "three", "other") # "three"
case(COUNT + 1, 4, "four", "other") # "other"
```

Booleans

A condition must be a boolean. There is no truthiness: a number, text or `null` where a condition is expected is an error, in if, not, `&&`, `||` and every other function that tests a condition. Write the test out: `COUNT > 0`, `TAG <> ""`, `X != null`.

To read one bit of a status word, bit gives a boolean, counting bits from `0`, the least significant.

```
if(COUNT > 0, "some", "none") # "some"
if(COUNT, "some", "none") # error
if(null, "some", "none") # error
not(PUMP_ON) # false
not(1) # error
1 && true # error
if(TAG <> "", TAG, "no tag") # "P-101"
bit(5, 0) # true
bit(5, 1) # false
```

Null

`null` is the absence of a value. It comes from the literal `null`, from if without an else branch when the condition does not hold, from case without a match or a default, from a field or a variable that holds `null`, and from the conversions given `null`.

- **Most operations refuse it.** `null` as a condition, as the right operand of an arithmetic operator, or as the argument of most functions is an error. `null` as the left operand of `+`, `-`, `*` or `/` gives `null`.
- **nvl replaces it**: `nvl(x, 0)` gives `x`, or `0` when `x` is `null`.

- **A variable that has never been written is not `null`**: it does not exist yet, and reading it is an error. See Errors and missing values.

```
if(COUNT > 5, "many") # null
case(COUNT, 1, "one") # null
{state: null}.state # null
nvl(if(COUNT > 5, "many"), "few") # "few"
to_string(null) # null
LEVEL + null # error
null + LEVEL # null
```

Lists and records

A list keeps its items in order, and may hold values of any kind. A record holds named fields, and **its fields have no guaranteed order**: a record is a hash map, so `{low: 10, high: 90}` may list `high` first. Read a record's fields by name, never by position. to_string writes a record's fields in alphabetical order, and so do the examples on these pages.

arrayAdd gives a new list with the values appended, and leaves the list it is given unchanged, as arrayConcat does. Before Amtiri Script 5.4.3 it added to the list it was given, so a variable's list changed behind the engine's back.

```
to_string(SETPOINTS) # "{high:90, low:10}"
length(SETPOINTS) # 2
arrayAdd([1, 2], 3) # [1, 2, 3]
with([1, 2], items, (arrayAdd(items, 3), items)) # [1, 2]
arrayConcat(REGISTERS, [7]) # [16256, 0, 65535, 1, 7]
```

Conversions

Function	Gives	From text	Fails on
to_int	A whole number; decimals are cut off	A whole number exactly as written: `"12"`, not `"12.7"` nor `" 12"`	Booleans, dates, lists, records
to_double	A decimal	A number with a point: `"1.5"`, `"1.5e3"`; spaces around it are ignored, a comma is not accepted	Booleans, dates, lists, records
to_string	The value written as text	—	Nothing
format	Text laid out by a pattern	—	A specifier that does not fit its value
type_of	The name of the value's kind	—	Nothing

`to_int`, `to_double`, `to_string` and `type_of` give `null` for `null`; `format` writes a `null` argument as the text `null`. A text that does not parse is an error: wrap the conversion in evl when the text comes from outside.

```
to_int(LEVEL) # 42
to_int("12") # 12
to_int("12.7") # error
to_int(" 12") # error
to_int(PUMP_ON) # error
to_double("1.5") # 1.5
to_double(" 1.5 ") # 1.5
to_double("1,5") # error
evl(to_double(TAG), 0.0) # 0.0
to_double(COUNT) # 3.0
to_string(LEVEL) # "42.5"
to_string(COUNT + 1) # "4.0"
to_string(REGISTERS) # "[16256, 0, 65535, 1]"
to_string(START) # "2026-09-29T08:00"
to_int(null) # null
```

Dates and time zones

A date and time carries no time zone. By convention it is in **UTC**: now gives UTC, and loggers stamp their samples in UTC.

- **Convert only to show.** toLocale reads a date and time as UTC and gives the same instant in the configured zone; toUTC goes the other way, and is what a date a person typed needs. The engine's zone is the instance's `application.timezone`. See the Configuration reference.
- **In a dashboard, the zone is the viewer's.** While a view redraws, the zone is the viewer's offset from UTC, as the browser gives it, in whole hours: a viewer at UTC+5:30 gets UTC+5. A variable formula that runs at that same moment may see that zone too, so convert in widget properties, not in variable formulas.
- **Build and take apart** with date, year, month, dayOfMonth and dayOfWeek, where Monday is `1`. date_of gives the start of the day.
- **Move with the add functions**, such as addHours. Days, hours, minutes and weeks may be fractions; months and years may not. `+` on a date and time gives `null`.
- **Compare** with the comparison operators, in time order.
- **Days as numbers.** toDays counts days from 1899-12-31, one less than a spreadsheet's serial number for any date since March 1900.
- **A dashboard's date field writes a date without a time of day.** `toLocale` and `toUTC` accept it as the start of that day; the other date functions need a date and time, so apply `toUTC` first.
- **Show a date** with format: `%tF` gives `2026-09-29`, `%tR` gives `08:00`, `%tT` gives `08:00:00`.

```
date(2026, 9, 29, 14, 30) # 2026-09-29T14:30
START.addHours(2) # 2026-09-29T10:00
START.addDays(1.5) # 2026-09-30T20:00
addMonths(date(2026, 1, 31), 1) # 2026-02-28T00:00
date_of(START) # 2026-09-29T00:00
year(START) # 2026
dayOfWeek(START) # 2
date(2026, 2, 30) # error
START + 1 # null
now() # *
toLocale(START) # *
toUTC(toLocale(START)) == START # true
"%tF %tR".format(START, START) # "2026-09-29 08:00"
```

Colours

A colour is a whole number that packs red, green and blue, as `0xRRGGBB`: pure red is `16711680`.

Function	Takes
rgb	Red, green and blue as fractions from `0` to `1`; `1` and above count as full
hsl	A hue in turns, from `0` to `1` (`0.3333` is green), then saturation and lightness from `0` to `1`
lighter, darker	A colour and a factor from `0` to `1`
gradient	The bare name of a gradient the engine provides, and a value from `0` to `1`

A widget's style needs a colour as text. Written as a number, the browser ignores it. Turn it into `#RRGGBB` with `"#%06X".format(colour)`:

```
rgb(1, 0, 0) # 16711680
"#%06X".format(rgb(1, 0, 0)) # "#FF0000"
"#%06X".format(rgb(0, 0.5, 1)) # "#0080FF"
"#%06X".format(hsl(0.3333, 1, 0.5)) # "#00FF00"
"#%06X".format(darker(rgb(1, 0, 0), 0.5)) # "#602020"
{fill: "#%06X".format(if(PUMP_ON, rgb(0, 0.5, 0), rgb(0.5, 0.5, 0.5)))} # {fill: "#008000"}
```

The engine provides the gradients `white2black`, `rainbow`, `black2green`, `black2red`, `red2gray2green`, `red2green` and `red2white2green`. A name it does not provide compiles, and fails when the formula runs, with `Unknown gradient 'NAME'`. The old spelling of the function, `gadient`, still works, but the palette offers only `gradient`.

```
{fill: "#%06X".format(gradient(red2green, LEVEL / 100))}
```

Next steps

- Errors and missing values
- Syntax

- Values and types
- Colours

This page describes Amtiri Script 5.4.3 as shipped with Data Orchester engine 6.18.2.