Expression parser 201 - All functions

Data types

General Information

Throughout the documentation we refer to **data types** that can be used in the **expression parser** and its functions.

The most commonly use data types are listed below.

> Most functions will accept **string** values so casting values to string is a very **powerful function**. Details can be found below in the **converting data types** section!
> Additionally you can directly transform a field value to text using the following syntax: **%{...anyfield}**

Data type	Description	Example
BOOLEAN	Comparison operators return a logical value true or false , as well as some functions.	**isActive**(string user_name)
NUMBER	This type represents numeric values, and is also used to store **Date**, **Time** and **Date-Time** (DATE_TIME) values. When storing any temporal value, the number represents the **milliseconds elapsed since January 1, 1970, 00:00:00 GMT**. Number or Date-Time fields can be referenced as numbers using the following notation: **{...somefield}**.	`1, 1.1, -1.1, .1, -.1`
STRING	This type represents any kind of text or character string including all kinds of select and multi-select fields MULTI . Any field type or data type is susceptible of being transformed to text, so **any field can be referenced as a text-string value** using the following notation: **%{...anyfield}**, and **%{...anyfield.i}** for Cascading Select or Multi-Cascading Select fields, where **i** is the index that represents the level to be accessed. (**i = 0** is used for base level).	`"Hello world"`
NUMBER []	This type represents a collection of numeric values returned by **various functions**. The size may vary from 0 to any number of numeric values. It is used to read the value of a numeric field in a selection of issues. You can also use literals like `[1, 2, 3]`.	`fieldValue(), append(), union(), except(), intersect() and distinct(), [1, 2, 3]`
STRING []	This type represents a collection of string values returned by **various functions**. The size may vary from 0 to any number of string values. It is also used to read the value of a string field in a selection of issues. You can also use literals like `["string_A", "string_B", "string_C"]`.	`fieldValue(), append(), union(), except(), intersect() and distinct(), ["string_A", "string_B", "string_C"]`

ISSUE []	This type represents a collection of issues. The size may vary from 0 to any number of issues. It's returned by issue selection or filtering functions like **subtasks(), linkedIssues(), filterByIssueType(), distinct()**, etc.	`subtasks()`, `linkedIssues()`, `transitionLinkedIssues()`, `filterByFieldValue()`, `filterByStatus()`, `filterByIssueType()`, `filterByResolution()`, `filterByProject()`, `append()`, `union()`, `except()`, `intersect()` and `distinct()`

Converting data types

There are multiple functions available for **converting** or **casting** data types. A comprehensive list can be found below.

Function	Input	Output
toString(number **n**)	NUMBER DATE_TIME	Returns a STRING with the decimal representation of the numeric value in **n**. Numeric value of a **Date-Time** field is number of milliseconds elapsed since January 1, 1970, 00:00:00 GMT. Example: `toString(3.141592)` returns `"3.141592"`.
toString(number **n**, number **decimals**)	NUMBER	Returns a STRING with the decimal representation of the numeric value in **n** limiting the fractional part to the number of digits in parameter **decimals**. Example: `toString(3.141592, 2)` returns `"3.14"` .
toString(number list **l**)	NUMBER []	Returns a STRING with a comma separated list of decimal representation of the numeric values in **l**. Example: `toString([1, 2, 3, 4.0])` returns `"1, 2, 3, 4"` .
toString(number list **l**, number **decimals**)	NUMBER []	Returns a STRING with a comma separated list of decimal representations of the numeric values in **l**, with the number of characters in the decimal part specified by parameter **decimals**. Example: `toString([1.123, 2.452, 3.64612], 2)` returns the following string: `"1.12, 2.45, 3.65"` .
toString(number list **l**, number **decimals**, string **separator**)	NUMBER []	Returns a STRING with a list of decimal representations of the numeric values in **l**, with the number of characters in the decimal part specified by parameter **decimals** and separated by string **separator**. Example: `toString([1.123, 2.452, 3.64612], 2, " : ")` returns the following string: `"1.12 : 2.45 : 3.65"` .
toString(string list **l**)	STRING []	Returns a STRING with a comma separated list of string values in **l**. Example: `toString(["Hello", " ", "world", "!"])` returns `"Hello, , world, !"` .

toString(string list **l**, string **separator**)	STRING []	Returns a STRING a list of string values in **l** separated by string **separator**. Example: **`toString(["blue", "red", "green"], "; ")`** returns "**`blue; red; green`**".
toString(issue list **l**)	ISSUE []	Returns a STRING with a comma separated list of issue keys. Example: **`toString(subtasks())`** returns "**`CRM-5, CRM-6`**" , being **CRM-5** and **CRM-6** the keys of current issue's **sub-tasks**.
toString(issue list **l**, string **separator**)	ISSUE []	Returns a STRING with a list of issue keys separated by string **separator**. Example: **`toString(subtasks(), " ")`** returns "**`CRM-5 CRM-6`**", being **CRM-5** and **CRM-6** the keys of current issue's **sub-tasks**.
toNumber(string **s**)	STRING	Returns the NUMBER represented by the string **s**. This function expects a decimal representation of a number. In case it is not possible to parse the **s** to number, null is returned. Example: **`toNumber("3.14")`** returns **`3.14`** .
toInteger(string **s**, string **radix**)	STRING	Returns the NUMBER represented by the string **s** as a signed integer in the radix specified by argument **radix**. Example: **`toInteger("ff", 16)`** returns **`255`** .
toStringList(string **s**, string **separators**)	STRING with a list of tokens separated by one or more characters	Returns a STRING [] with tokens in argument **s** separated by characters in argument **separators**. Leading and trailing spaces around each token are automatically removed. Example: **`toStringList("red, orange, yellow; green; blue; purple", ",;")`** returns the following string list: **`["red", "orange", "yellow", "green", "blue", "purple"]`** .
toStringList(multi-valued field **field**)	field code for a MULTI -value field in format **%{...somefield}**. Multi-valued fields are Multi Select, Checkboxes, Components, Versions, Multi User Picker, Multi Group Picker, Issue Pickers, Attachments and Labels.	Returns a STRING [] representing each of the values selected in the **field**. Example: **`toStringList(%{...components})`** returns a list of strings with each of the components selected in current issue.
toNumberList(string **s**, string **separators**)	STRING with a list of numbers in decimal representation separated by one or more characters	This function expects in argument **s** a string containing numbers in decimal representation separated by characters in argument **separators**, and returns a NUMBER []. Example: **`toNumberList("1, 3, 5; 7; 11; 13", ",;")`** returns the following number list: **`[1, 3, 5, 7, 11, 13]`** .

issueKeysToIssueList(string **issue_keys**)	STRING with a comma separated list of issue keys	Returns an ISSUE [] with all issues with keys in argument **issue_keys**. Argument **issue_keys** is a string containing a comma separated list of issue keys. Example: **`issueKeysToIssueList("CRM-12, HT-254")`** returns an issue list with issues with keys **CRM-12** and **HT-254**.

Automatic casting from Number to Text-String

Whenever you write a numeric term at the right-hand side of **concat operator +** or a **comparison operator** like **=** , and the left-hand side is occupied by a text-string term, the parser will automatically transform the right-hand side term into a string

- **+ (string concat)**: **`"His age is " + 30`** is equivalent to **`"His age is " + toString(30)`**
- **= (any comparison operator)**: **`"30" = 30`** is equivalent to **`"30" = toString(30)`**

Operators

General Information

The expression parser accepts the most common operators. The operators listed below are available for the following **data types**:

- **Numbers**
- **Strings**
- **Issue lists**
- **Number lists**
- **String lists**

Operators = and != are also available for type BOOLEAN

Case-sensitive operators

Operator	Meaning	Examples (all examples return `true`)
=	equal to	**`1 = 1`** **`"HELLO" = toUpperCase("Hello")`** **`%{...description} = {...timeoriginalestimate}`** , auto-casting numeric field **{...originalEstimate}** to Text-String. **`%{...originalEstimate} = toString({...originalEstimate})`** , explicit casting of numeric field **{...originalEstimate}** to Text-String. **`true = true`** **`%{...cf10001} = null`** , for checking whether field with code **%{...cf10001}** is not initialized. **`[1, 2, 3] = [1, 2, 3]`** , when used with lists elements existence and its order are evaluated. **`["blue", "red", "green"] = ["blue", "red", "green"]`**
!=	not equal to	**`0 != 1`** **`"HELLO" != "Hello"`** **`%{...description} != "Hello"`** **`true != false`** **`{...cf10010} != null`** , for checking whether the numeric field with code **{...cf10010}** is initialized. **`[1, 2, 3] != [1, 3, 2]`** , when used with lists elements existence and its order are evaluated. **`["blue", "red", "green"] != ["blue", "green", "red"]`**

<	lower than	**`1 < 2`** **`"abc" < "bbc"`** **`"abc" < "abcd"`**
>	greater than	**`2 > 1`** **`"bbc" > "abc"`** **`"abcd" > "abc"`**
<=	less than or equal to	-
>=	greater than or equal to	-
~	contains	**`"Hello world!" ~ "world"`** , checks whether a string contains a substring. **`%{...componentLeads} ~ %{...currentUser}`** , checks whether "**Component leaders**" contains "**Current user**". **`linkedIssues() ~ subtasks()`** , checks whether all sub-tasks are also linked to current issue. **`[1, 2, 3, 2, 2, 4] ~ [2, 1, 2]`** , when used with lists cardinalities must match. **`["blue", "red", "green", "red", "white", "red"] ~ ["red", "green", "red"]`** **`(["green", "red"] ~ ["red", "green", "red"]) = false`**
!~	doesn't contain	**`"world" !~ "Hello world!"`** **`%{...fixVersions} !~ %{...versions}`** , checks whether "**Fix version/s**" doesn't contain all versions in "**Affects version/s**". **`fieldValue(%{...reporter}, linkedIssues()) !~ fieldValue(%{...reporter}, subtasks())`** , checks whether linked issues reporters don't include all sub-tasks reporters. **`[1, 2, 3, 2, 2, 4] !~ [2, 1, 1, 4]`** , when used with lists cardinalities must match. **`["blue", "red", "green", "red", "red"] !~ ["red", "green", "green", "red"]`**
in	is contained in	**`"world" in "Hello world!"`** , to check whether a substring is contained in a string. **`%{...currentUser} in %{...componentLeads}`** , checks whether "**Current user**" is contained in "**Component leaders**". **`subtasks() in linkedIssues()`** , checks whether all sub-tasks are also linked to current issue. **`[1, 1, 2] in [2, 1, 1, 1, 4]`** , cardinality must match. **`["blue", "red", "red"] in ["red", "green", "blue", "red", "red"]`** , cardinality must match. **`2 in [1, 2, 3]`** **`"blue" in ["red, "blue", "white"]`**
not in	isn't contained in	**`"Hello world!" not in "world"`** **`%{...versions} not in %{...fixVersions}`** , checks whether not all versions in "**Affects version/s**" are contained in "**Fix version/s**". **`fieldValue(%{...reporter}, subtasks()) not in fieldValue(%{...reporter}, linkedIssues())`** , checks whether not all sub-tasks reporters are included in linked issues reporters. **`[1, 1, 2, 2] not in [2, 1, 1, 1, 4]`** , cardinality must match. **`["blue", "red", "red", "blue"] not in ["red", "blue", "red", "red"]`** , cardinality must match. **`5 not in [1, 2, 3, 3, 4]`** **`"orange" not in ["blue", "red", "white"]`**
any in	some element is in	**`%{...versions} any in %{...fixVersions}`** , checks whether any version in "**Affects version/s**" is contained in "**Fix version/s**". **`fieldValue(%{...reporter}, subtasks()) any in fieldValue(%{...reporter}, linkedIssues())`** , checks whether any sub-task's reporter is present among linked issues reporters. **`[1, 3] any in [3, 4, 5]`** **`["blue", "white"] any in ["black", "white", "green"]`**
none in	no single element is in	**`%{...versions} none in %{...fixVersions}`** , checks whether there isn't a single version "**Affects version/s**" in "**Fix version/s**". **`fieldValue(%{...reporter}, subtasks()) none in fieldValue(%{...reporter}, linkedIssues())`** , checks whether there isn't a single sub-task reporter among linked issues reporters. **`[1, 2] none in [3, 4, 5]`** **`["blue", "red"] none in ["black", "white", "green"]`**

Case-ignoring Operators

The following comparison operators are applicable to STRING and STRING [] **data types**.

All operators ignore the case of the characters.

Operator	Meaning	Examples (all examples return `true`)

`=~`	equal to	`"HELLO" =~ "Hello"`
		`"up" =~ "UP"`
		`["blue", "red", "green"] =~ ["Blue", "RED", "Green"]`

`!=~`	not equal to	`" HELLO" !=~ "Hello"`
		`"up" !=~ "down"`
		`("up" !=~ "UP") = false`
		`["blue", "red"] !=~ ["Blue", "green"]`
		`["blue", "red"] !=~ ["Red", "BLUE"]`
		`(["blue", "red", "green"] !=~ ["Blue", "RED", "Green"]) = false`

`~~`	contains	`"Hello World!" ~~ "world"` , checks whether a string contains a substring. `"A small step for a man" ~~ "STEP"` , checks whether a string contains a substring. `["one", "two", "three"] ~~ ["TWO", "One"]` , checks whether a string list contains all the elements of another string list.
`!~~`	doesn't contain	`"Hello World!" !~~ "bye"` , checks whether a string doesn't contain a substring. `"A small step for a man" !~~ "big"` , checks whether a string doesn't contain a substring. `["one", "two", "three"] !~~ ["Four"]` , checks whether a string list doesn't contain one element of another string list. `(["one", "two", "three"] !~~ ["TWO"]) = false`
`in~`	is contained in	`"world" in~ "Hello World!"` , checks whether a substring is contained in another string. `"STEP" in~ "A small step for a man"` , checks whether a substring is contained in another string. `["TWO", "One"] in~ ["one", "two", "three"]` , checks whether all the elements of a string list are contained in another string list.
`not in~`	isn't contained in	`"bye" not in~ "Hello World!"` , checks whether a substring is not contained in another string. `"big" not in~ "A small step for a man"` , checks whether a substring is not contained in another string. `["Four"] not in~ ["one", "two", "three"]` , checks whether any of the elements of a string list are not contained in another string list. `(["TWO"] not in~ ["one", "two", "three"]) = false`
`any in~`	some element is in	`["blue", "violet"] any in~ ["Blue", "Red", "Green"]` `["Five", "One"] any in~ ["FOUR", "FIVE", "SIX"]`
`none in~`	no single element is in	`["Orange"] any in~ ["red", "blue", "green"]` `(["orange"] any in~ ["Red", "Orange"]) = false`

Operators and applicable data types

Below you find a comprehensive matrix of all operators and applicable data types.

Comparison Operator	BOOLEAN	NUMBER	STRING	NUMBER []	STRING []	ISSUE
`=`	X	X	X	X	X	X
`!=`	X	X	X	X	X	X
`<`	-	X	X	-	-	-
`>`	-	X	X	-	-	-
`<=`	-	X	X	-	-	-
`>=`	-	X	X	-	-	-

~	-	-	X	X	X	X
!~	-	-	X	X	X	X
in	-	-	X	X	X	X
not in	-	-	X	X	X	X
any in	-	-	-	X	X	X
none in	-	-	-	X	X	X
=~	-	-	X	-	X	-
!=~	-	-	X	-	X	-
~~	-	-	X	-	X	-
!~~	-	-	X	-	X	-
in~	-	-	X	-	X	-
not in~	-	-	X	-	X	-
any in~	-	-	-	-	X	-
none in~	-	-	-	-	X	-

Remember	Example
Operators **~, !~, in** and **not in** can be used for checking a single element **(number or string)** against a **number list** or a **string list**	• **1 in [1, 2, 3]** • **["blue", "red"] ~ "blue" .**
Operators **~, !~, in** and **not in** when used with a **string** are useful to look for substrings in another string.	• **"I love coding" ~ "love"** • **"I don't like Mondays" !~ "Fridays"** • **"love" in "I love coding"** • **"Fridays" not in "I don't like Mondays".**
Operators **~, !~, in** and **not in** respect cardinality, i.e., container list must have at least the same number of elements as contained list.	• **[1, 1] in [1, 1, 1]** • **[1, 1] not in [1, 2, 3] .**
Operators **=** and **!=**, when used for comparing lists, require to have the **same elements**, with the **same cardinality** and the **same order**.	• **[1, 2, 3] = [1, 2, 3]** • **[4, 5, 6] != [4, 6, 5] .**
Operators **<, >, <=** and **>=** work according to lexicographical order when comparing strings.	

A reference of all data types can be found here.

Boolean expressions

Fixed values

Only two values will be accepted / returned: **`true`** and **`false`**.

Logical operators

The following logical operators can be used for linking logical terms in an expression, i.e., terms that return a boolean value type (**true** or **false**).

Operator	Meaning	Precedence
`NOT or !`	logical negation	1 (highest)
`AND` or **`&`**	logical conjunction	2
`OR or |`	logical disjunction	3
`XOR`	exclusive or, i.e., **`a XOR b`** is equivalent to **`a AND !b OR !a AND b`**	3
`IMPLIES or IMP`	logical implication, i.e., **`a IMPLIES b`** is equivalent to **`!a OR b`**	4
`XNOR or EQV`	logical equivalence, i.e., **`a EQV b`** is equivalent to **`a IMPLIES b AND b IMPLIES a`**	4 (lowest)

Logical connectives are case insensitive, i.e., they can also be written in lower case: **`or, and, not, xor, implies, imp, eqv`** and **`xnor`**.

Conditional operator: ? **:** (IF, THEN, ELSE)

The conditional operator **`? :`** is a powerful operator to construct conditional expressions.

The conditional operator basically allows you to construct the following expression: **IF** boolean _expression **true THEN** term_1 **ELSE** term_2.

The format to be used is: <boolean_expression> ? <term_1> : <term_2>

Both **term_1** and **term_2** need to be of the same **data type** (**boolean, number, string, issue list, string list** or **number list**).

Examples of using the conditional operator

Expression	Output
`{...duedate} != null ? ({...duedate} - {...currentDateTime}) / {HOUR} : 0`	**If** the **Due Date** is **`not null`**, this function will return the number of **hours from the current date-time** to **Due Date**, otherwise it will return **`0`** .
`timePart({...currentDateTime}, LOCAL) > 21:00 AND timePart({...currentDateTime}, LOCAL) < 7:00 ? "Night" : "Day"`	**If** the current time is **between 21:00 and 7:00** this function will return "**`Night`**" , otherwise it will return "**`Day`**".

Examples

Input	Output
`%{...somefield} = "Yes"`	**`True`** if the value of the field is "Yes", otherwise **`False`**.
`%{...somefield1} != null AND %{...somefield2} = null`	**`True`** only if **{...somefield1}** field **has a value** and field **{...somefield2}** does **NOT have a value**.
`datePart({...duedate}, LOCAL) > datePart({...currentDateTime}, LOCAL)`	**`True`** only if **Due Date** (field code **{...duedate}**) is later than **Current date** (field code **{...currentDateTime}**) in server's local timezone.

Numbers, dates and times

Fixed values

Input	Format	Example
Valid **numerical** values		• `1` • `3.0` • `.5` • ` -400` • `-1.1` • ` -.02`
Valid **Date-time** values	**yyyy/MM/dd [hh:mm]**or **yyyy-MM-dd [hh:mm]**	• `2018/03/25 23:15` • `2018-03-25 23:15` • `2018/03/25` • `2018-03-25`
Valid **Time** values	**hh:mm**	• `08:15` • `23:59` • `00:00`

Variable values (field values)

Numeric values of **Number**, **Date**, **Date-Time** and **Priority** data type fields can be inserted in expressions with following notation **{...somefield}**, e.g., use **`{...duedate}`** for **Due Date**, and **`{...numberOfAttachments}`** for **Number of attachments**.

Pro tip

For checking if a field is has a value you can use **`{...somefield} = null`** or **`{...somefield} != null`**

Math Functions

Function	Input	Returned value
abs(number **x**)	NUMBER	Returns the absolute value of **x**, i.e., if x>0 it returns **x**, otherwise it returns **-x**.
acos(number **x**)	NUMBER	Returns the arc cosine of **x**; the returned angle is in the range 0.0 through pi.
asin(number **x**)	NUMBER	Returns the arc sine of **x**; the returned angle is in the range 0.0 through pi.
atan(number **x**)	NUMBER	Returns the arc tangent of **x**; the returned angle is in the range 0.0 through pi.
ceil(number **x**)	NUMBER	Returns the smallest (closest to negative infinity) value that is larger than or equal to **x** and is equal to a mathematical integer.
cbrt(number **x**)	NUMBER	Returns the cube root of **x**.
cos(number **x**)	NUMBER	Returns the trigonometric cosine of angle **x** expressed in radians.
cosh(number **x**)	NUMBER	Returns the hyperbolic cosine of **x**.
floor(number **x**)	NUMBER	Returns the largest (closest to positive infinity) value that is less than or equal to **x** and is equal to a mathematical integer.

log(number **x**)	NUMBER	Returns the natural logarithm (base e) of **x**.
log10(number **x**)	NUMBER	Returns the base 10 logarithm of **x**.
max(number **x**, number **y**)	NUMBER	Returns the larger of two numeric values.
min(number **x**, number **y**)	NUMBER	Returns the smaller of two numeric values.
modulus(number **divi dend**, number **divisor**)	NUMBER	Returns `dividend - (divisor * floor (dividend / divisor))` .
pow(number **x**, number **y**)	NUMBER	Returns **x** raised to the power **y**.
random()	NUMBER	Returns a value with a positive sign, greater than or equal to 0.0 and less than 1.0.
remainder(number **div idend**, number **divisor**)	NUMBER	Returns `dividend - divisor * n` , where **n** is the closest integer to `dividend / divisor` .
round(number **x**)	NUMBER	Returns the closest integer to **x**.
sin(number **x**)	NUMBER	Returns the trigonometric sine of angle **x** expressed in radians.
sinh(number **x**)	NUMBER	Returns the hyperbolic sine of **x**.
sqrt(number **x**)	NUMBER	Returns the square root of **x**.
tan(number **x**)	NUMBER	Returns the trigonometric tangent of angle **x** expresse d in radians.
tanh(number **x**)	NUMBER	Returns the hyperbolic tangent of **x**.
toDegrees(number **x**)	NUMBER	Converts an angle **x** measured in radians to an approximately equivalent angle measured in degrees.
toRadians(number **x**)	NUMBER	Converts an angle **x** measured in degrees to an approximately equivalent angle measured in radians.

Date-Time Functions

Fields of type **Date** and **Date-Time** contain a **numeric value** with the **milliseconds elapsed since January 1, 1970, 00:00:00 GMT**. We usually need to get significative numbers from this numeric value, like YEAR, MONTH, DAY, HOUR, MINUTE, etc.

To achieve this, **Automation Toolbox for Jira** provides a comprehensive set of functions, most of them with **TIMEZONE** as input argument, since any significative number relative to a timestamp depends on the timezone.

Function	Input	Returned value
timePart(number **t**, timeZone **time_zone**)	TIMEZONE	Returns the time part of timestamp represented by numeric value **t** in **time_zone** time zone. Example: for timestamp **March, 25th 2011 23:15** this function returns a NUMBER representing time **23:15** in milliseconds
datePart(number **t**, timeZone **time_zone**)	TIMEZONE	Returns the date part of timestamp represented by numeric value **t** in **time_zone** time zone. Example: for timestamp **March, 25th 2011 23:15** this function returns a NUMBER representing date **March, 25th 2011 00:00** in milliseconds

second(number **t**, timeZone **time_zone**)	TIMEZONE	Returns the seconds figure of timestamp represented by numeric value **t** in **time_zone** time zone. Example: for timestamp **March, 25th 2011 23:15:30** this function returns a NUMBER representing **30 seconds** in milliseconds.
minute(number **t**, timeZone **time_zone**)	TIMEZONE	Returns the minutes figure of timestamp represented by numeric value **t** in **time_zone** time zone. Example: for timestamp **March, 25th 2011 23:15:30** this function returns a NUMBER representing **15 minutes** in milliseconds.
hour(number **t**, timeZone **time_zone**)	TIMEZONE	Returns the hours figure of timestamp represented by numeric value **t** in **time_zone** time zone. Example: for timestamp **March, 25th 2011 23:15:30** this function returns a NUMBER representing **23 hours** in milliseconds.
dayOfTheWeek(number **t**, timeZone **time_zone**)	NUMBER	Returns the day of the week of timestamp represented by numeric value **t** in **time_zone** time zone, with Sunday = 1, Monday = 2, ... Saturday = 7. Example: for timestamp **March, 25th 2011 23:15** this function returns **6** for Friday as a NUMBER, represented also by macro `{FRIDAY}` .
dayOfTheMonth(number **t**, timeZone **time_zone**)	NUMBER	Returns the day of the month of timestamp represented by numeric value **t** in **time_zone** time zone. Example: for timestamp **March, 25th 2011 23:15** this function returns **25** as a NUMBER.
month(number **t**, timeZone **time_zone**)	NUMBER	Returns the month of a timestamp represented by numeric value **t** in a certain time zone, with January = 1, February = 2, ... December = 12. Example: for timestamp **March, 25th 2011 23:15** this function returns **3** for March as a NUMBER, represented also by macro `{MARCH}` .
year(number **t**, timeZone **time_zone**)	NUMBER	Returns the year of a timestamp represented by numeric value **t** in a certain time zone. Example: for timestamp **March, 25th 2011 23:15** this function returns **2011** as a NUMBER .
addDays (number **t** , number **n**, timeZone **time_zone**)	NUMBER	Returns a timestamp as a NUMBER resultant of adding **n** days to timestamp **t**. You should use this function instead of simply adding `n * {DAY}` , since `{DAY}` is a macro equivalent to `24 * {HOUR}` , not taking into account that once in a year we have a day with 25 or 23 hours due to DST transition. Negative values for `n` are used in order to subtract instead of adding. Example: `addDays(2018/03/27 01:00, -2, LOCAL)` returns **2018/03/25 01:00** .
addMonths(number **t**, number **n**, timeZone **time_zone**)	NUMBER	Returns a timestamp resultant of adding **n** months to timestamp **t**. You should use this function instead of simply adding `n * {MONTH}` , since `{MONTH}` is a macro equivalent to `30 * {DAY}` , not taking into account that some months has more or less than 30 days. Negative values for **n** are used in order to subtract instead of adding. Example: for timestamp **t** with value **March, 25th 2011 23:15** calling to `addMonths(t, 3, LOCAL)` will return a timestamp as a NUMBER with value **June, 25th 2011 23:15**

addYears(number **t**, number **n**, timeZone **time_zone**)	NUMBER	Returns a timestamp resultant of adding **n** years to timestamp **t**. You should use this function instead of simply adding **`12 * {MONTH}`** or **`365 * {DAY}`** , since that won't take into account that some years have 366 days. Negative values for **n** are used in order to subtract instead of adding. Example: for timestamp **t** with value **March, 25th 2011 23:15** calling to **`addYears(t, 10, LOCAL)`** will return a timestamp as a NUMBER with value **March, 25th 2021 23:15**
addTimeSkippingWeekends(number **t** , number **timeToBeAdded**, timeZone **time_zone**)	NUMBER	Adds **timeToBeAdded** to **t** and returns a NUMBER with the difference that weekends don't count in the sum, e.g., if **t** represents a date-time which coincides with a Saturday, adding **timeToBeAdded = 2 * {HOUR}** will return a **date-time for next Monday at 02:00** . Use negative values at **timeToBeAdded** for subtracting time from **t**.
addTimeSkippingWeekends(number **t** , number **timeToBeAdded**, timeZone **time_zone**, number **beginning_of_weekend**, number **end_of_weekend**)	NUMBER	Same as previous function, returns a NUMBER but with a custom defined weekend. Arguments **beginning_of_weekend** and **end_of_weekend** take values **`{MONDAY}`** , **`{TUESDAY} ... {SUNDAY}`** . Example of usage for adding 12 hours to **Current date and time** using Israeli weekend: **`addTimeSkippingWeekends({...currentDateTime}, 12 * {HOUR}, LOCAL, {FRIDAY}, {SATURDAY})`**
addDaysSkippingWeekends(number **t** , number **n**, timeZone **time_zone**)	NUMBER	Returns a timestamp as a NUMBER equivalent of **`t + n*{DAY}`** with the difference that weekends don't count in the sum, e.g., if **t** represents a timestamp which coincides with a Friday, adding n = 1 will return a date-time for next Monday. Negative values for **n** are used in order to subtract days to **t**.
addDaysSkippingWeekends(number **t** , number **n**, timeZone **time_zone** , number **beginning_of_weekend**, number **end_of_weekend**)	NUMBER	Same as previous function, returns a NUMBER but with a custom defined weekend. Arguments **beginning_of_weekend** and **end_of_weekend** take values **`{MONDAY}, {TUESDAY} ... {SUNDAY}`** . Example of usage for adding 10 workdays to **Due date** using Israeli weekend: **`addDaysSkippingWeekends({...duedate}, 10, LOCAL, {FRIDAY}, {SATURDAY})`**
subtractDatesSkippingWeekends(number **minuend_date** , number **subtrahend_date**, timeZone **time_zone**)	NUMBER	Returns a timestamp as a NUMBER equivalent "**minuend_date - subtrahend_date**" subtracting weekend periods from the result, i.e., you get the elapsed working time from **subtrahend_date** to **minuend_date**.
subtractDatesSkippingWeekends(number **minuend_date** , number **subtrahend_date**, timeZone **time_zone**, number **beginning_of_weekend**, number **end_of_weekend**)	NUMBER	Same as previous function, returns a NUMBER but with a custom defined weekend. Arguments **beginning_of_weekend** and **end_of_weekend** take values **`{MONDAY}, {TUESDAY} ... {SUNDAY}`** . Example of usage calculating the worktime from **Creation** to **Resolution** using Israeli weekend: **`subtractDatesSkippingWeekends({...resolutiondate}, {...created}, LOCAL, {FRIDAY}, {SATURDAY})`**
dateToString(number **t**, timeZone **time_zone**, **language**)	NUMBER	Returns a STRING representing the date-time value at **t**, in a certain **time zone**, and in a certain **language**. This function is useful in the **Action Update Field** to represent as a string the result of a time expression.
dateTimeToString(number **t**, timeZone **time_zone**, **language**)	NUMBER	Returns a STRING representing the date-time value at **t**, in a certain **time zone**, and in a certain **language**. This function is useful in the **Action Update Field** to represent as a string the result of a time expression.

dateTimeToString(number **t**, string **date_time_pattern** , **language**)	NUMBER	Returns a STRING representing the date-time value at **t** with a certain custom format defined by **date_time_pattern** string parameter, using a certain **language** when using words for months, days of the week, etc. This function is useful in **Action Update Field** to represent as a string the result of a time expression. Example: **`dateTimeToString(2011-03-25 11:30, "yyyy.MM.dd 'at' HH:mm:ss", USER_LANG)`** returns string "**2011.03.25 at 11:30:00**".
dateTimeToString(number **t**, string **date_time_pattern** , timeZone **time_zone** , **language**)	NUMBER	Returns a STRING representing the date-time value at **t** with a certain custom format defined by **date_time_pattern** string parameter, in a certain timezone **time_zone**, using a certain **language** when using words for months, days of the week, etc. This function is useful in the **Action Update Field** to represent as a string the result of a time expression. Example: **`dateTimeToString(0, "yyyy.MM.dd 'at' HH:mm:ss", GMT, USER_LANG)`** returns string "**1970.01.01 at 00:00:00**". Example: **`dateTimeToString(0, "yyyy.MM.dd 'at' HH:mm:ss", MST, USER_LANG)`** returns string "**1969.12.31 at 17:00:00**".
monthToString(number **t**, timeZone **time_zone**, **language**)	NUMBER	Returns a STRING with the name of the month for a date-time **t**, in a certain time zone **time_zone** , and in a certain **language**. This function can be used in the **Action Update Field** to write the name of the month of a date-time field or expression.
dayOfTheWeekToString(number **t**, timeZone **time_zone** , **language**)	NUMBER	Returns a STRING with the day of the week for a date-time **t**, in a certain time zone **time_zone**, and in a certain **language**. This function is useful in the **Action Update Field** to write the day of the week of a date-time field or expression.
weekOfTheYear(number **t**, number **firstDayOfTheWeek**, number **minimalDaysInFirstWeek**, timeZone **time_zone**) Available since version 1.1.0	NUMBER NUMBER NUMBER TIMEZONE	Returns the **week of the year** of the date-time t in a certain **time_zone** as NUMBER. The parameter **firstDayOfTheWeek** represents the first day of the week, e.g.: **{SUNDAY}** in the U.S., and **{MONDAY}** in Germany. The parameter **minimalDaysInFirstWeek** represents the minimal number of days required in the first week of the year, e.g., if the first week is defined as the one that contains the first day of the first month of the year, value **1** should be used. If the minimal number of days required must be a full week (e.g. all days of the week need to be in that year), value **7** should be used. Example: **`weekOfTheYear(2023/01/03, {SUNDAY}, 1, LOCAL)`** returns **1**. Example: **`weekOfTheYear(2023/01/03, {MONDAY}, 1, LOCAL)`** returns **2**. Example: **`weekOfTheYear(2023/01/03, {MONDAY}, 7, LOCAL)`** returns **1**.
dayOfTheYear(number **t**, timeZone **time_zone**) Available since version 1.1.0	NUMBER TIMEZONE	Returns the **day of the year** of date-time **t** in a certain **time_zone** as NUMBER, e.g. for January 1st the value returned will be **1**. Example: **`dayOfTheYear(2019/02/01, LOCAL)`** returns **32**
stringToDate(string **s** , timeZone **time_zone**)	TIMEZONE STRING	Returns a NUMBER with the date-time represented by string **s**. The numeric value returned corresponds to the milliseconds elapsed since **January 1, 1970, 00:00:00 GMT**. Valid input string formats are **yyyy/MM/dd HH:mm, yyyy-MM-dd HH:mm, yyyy/MM/dd, yyyy-MM-dd**, also formats relative to current time like in JQL queries: "**w**" (weeks), "**d**" (days), "**h**" (hours) or "m**"** (minutes), or format defined at system property **jira.date.time.picker.java.format**.

stringToDate(string **s** , string **date_time_pattern**)	STRING	Returns a NUMBER with the date-time represented by string **s**. Expected format of value at parameter "**s**" is defined by **date_time_pattern** string parameter. The numeric value returned corresponds to the milliseconds elapsed since **January 1, 1970, 00:00:00 GMT**. Example: **`stringToDate("2011.03.25 at 11:30:00", "yyyy.MM.dd 'at' HH:mm:ss")`** returns a date-time numeric value that can be used for setting a Date Time picker custom field.
stringToDate(string **s** , string **date_time_pattern** , string **language** , string **country**)	STRING	Returns a NUMBER with the date-time represented by string **s**. Expected format of value at parameter "**s**" is defined by **date_time_pattern** string parameter for a specific **language** (language code ISO 639-2) and **country** (country code ISO 3166 alpha-2). The numeric value returned corresponds to the milliseconds elapsed since **January 1, 1970, 00:00:00 GMT**. Example: **`stringToDate("Dec 7, 2016 2:10:25 AM PST", "MMM d, yyyy h:mm:ss a z", "eng", "US")`** returns a date-time numeric value that can be used for setting a Date Time picker custom field.
formatDuration(number **duration**)	DURATION	Returns a STRING with the pretty representation of a time **duration**, i.e. a subtraction of 2 date-time values, using the language of current user's profile. Example: **`formatDuration(2017-01-31 11:30 - 2017-01-30 00:00)`** returns "**1 day, 11 hours, 30 minutes**" .
shortFormatDuration(number **duration**)	DURATION	Returns a STRING with the most compact representation possible of a time **duration**, i.e. a subtraction of 2 date-time values, using the language of current user's profile. Example: **`shortFormatDuration(2017-01-31 11:30 - 2017-01-30 00:00)`** returns "**1d 11h 30m**" .
formatWorkDuration(number **duration**)	DURATION	Returns a STRING similar to function **`formatDuration()`** but using the workday and workweek defined at **time tracking configuration** , instead of 24 hours per day and 7 days per week. Example: **`formatWorkDuration(5 * 8 * {HOUR} + 2 * 8 * {HOUR} + 3 * {HOUR})`** returns "**1 week, 2 days, 3 hours**", with 8 hours per **workday** and 5 days per **workweek**.
shortFormatWorkDuration(number **duration**)	DURATION	Returns a STRING similar to function **`shortFormatDuration()`** but using the workday and workweek defined at **time tracking configuration** , instead of 24 hours per day and 7 days per week. Example: **`formatWorkDuration(5 * 8 * {HOUR} + 2 * 8 * {HOUR} + 3 * {HOUR})`** returns "**1w 2d 3h**" , with 8 hours per **workday** and 5 days per **workweek** .
timeZone(string **timeZone_name**)	TIMEZONE	Returns the timeZone whose name is represented by string **timeZone_name**. This function is useful to obtain a **timeZone** from a string, like the value of a **Project Properties**. Example: **`timeZone("DST")`** returns **DST timeZone**.

timeInValue(string field **field**, boolean expression **predicate**) Available since version 1.1.0	STRING BOOLEAN	Returns the NUMBER of milliseconds a string **field** with code **%{nnnnn}** of the current issue has had a value satisfying a boolean expression **predicate**, where the string value of the **field** with code **%{nnnnn}** is represented by **^%**. Example: **`timeInValue(%{00000}, ^% ~~ "ERROR" OR ^% ~~ "WARNING")`** returns the number of milliseconds the field summary (field code **%{00000}**) of the current issue has contained any of the words "ERROR" or "WARNING", ignoring the case. Example: **`timeInValue(%{00094}, count(toStringList(^%, ",")) > 1)`** returns the number of milliseconds the field components (field code **%{00094}**) of the current issue has contained more than one selected component. Example: **timeInValue(%{00017}, ^% in ["Critical", "High"])** returns the number of milliseconds the field priority (field code **%{00017}**) of the current issue has had a value of Critical or High.
timeInValue(number field **field**, boolean expression **predicate**) Available since version 1.1.0	NUMBER BOOLEAN	Returns the NUMBER of milliseconds a number or date-time **field** with code **{nnnnn}** of the current issue has had a value satisfying a boolean expression **predicate**, where the numeric value of the **field** with code **{nnnnn}** is represented by **^**. Example: **`timeInValue({00012}, ^ != null)`** returns the number of milliseconds the field Due date (field code **{00012}**) of the current issue has had a value. Example: **`timeInValue({10001}, ^ >= 5 AND ^ <= 10)`** returns the number of milliseconds a hypothetical numeric field called Passengers (field code **{10001}**) of the current issue has remained between 5 and 10. Example: **`timeInValue({10001}, modulus(^, 2) = 0)`** returns the number of milliseconds a hypothetical numeric field called Passengers (field code **{10001}**) of the current issue has had an even value (2, 4, 6,...).
timeInValue(string field **field**, issue list **issues**, boolean expression **predicate**) Available since version 1.1.0	STRING ISSUE [] BOOLEAN	Returns the sum of milliseconds a string **field** with code **%{nnnnn}** has had a value satisfying a boolean expression **predicate** in distinct issues as NUMBER, where the string value of the field with code **%{nnnnn}** is represented by **^%**. Example: **`timeInValue(%{00000}, subtasks(), ^% ~~ "ERROR" OR ^% ~~ "WARNING")`** returns the sum of milliseconds the summary fields (field code **%{00000}**) of all subtasks of the current issue have contained any of the words "ERROR" or "WARNING", ignoring the case. Example: **`timeInValue(%{00094}, epic(), count(toStringList(^%, ",")) > 1)`** returns the number of milliseconds the components fields (field code **%{00094}**) in a linked Epic issue have contained more than one selected component. Example: **`timeInValue(%{00017}, filterByIssueType(linkedIssues(), "Bug, New Feature"), ^% in ["Critical", "High"])`** returns the sum of milliseconds all linked Bugs and New Features of the current issue have had a priority (field code **%{00017}**) value of Critical or High.

timeInValue(number field **field**, issue list **issues**, boolean expression **predicate**) Available since version 1.1.0	NUMBER ISSUE [] BOOLEAN	Returns the sum of milliseconds a number or date-time **field** with code **{nnnnn}** has had a value satisfying a boolean expression **predicate** in distinct issues as NUMBER, where the numeric value of the **field** with code **{nnnnn}** is represented by **^**. Example: **`timeInValue({00012}, subtasks(), ^ != null)`** returns the number of milliseconds the field due date (field code **{00012}**) of all subtasks of the current issue have had a value. Example: **`timeInValue({10001}, epic(), ^ >= 5 AND ^ <= 10)`** returns the number of milliseconds a hypothetical numeric field called Passengers (field code **{10001}**) of an Epic issue has had a value between 5 and 10. Example: **`timeInValue({10001}, filterByIssueType(linkedIssues(), "Bug, New Feature"), modulus(^, 2) = 0)`** returns the number of milliseconds a hypothetical numeric field called Passengers (field code **{10001}**) has had an even value in any linked Bug or New Feature.
timeInValue(string field **field**, boolean expression **predicate** , string **schedule_name**, timeZone **time_zone**) Available since version 1.1.0	STRING BOOLEAN STRING TIMEZONE	Returns the NUMBER of milliseconds a string **field** with code **%{nnnnn}** of the current issue has had a value satisfying a boolean expression **predicate**, where the string value of the **field** with code **%{nnnnn}** is represented by **^%**. The time being calculated by this function is only counted during a defined schedule with name **schedule_name** for time zone **time_zone**. Example: **`timeInValue(%{00000}, ^% ~~ "ERROR" OR ^% ~~ "WARNING", "schedule_name", LOCAL)`** returns the number of milliseconds the field summary (field code **%{00000}**) of the current issue has contained any of the words "ERROR" or "WARNING", ignoring the case, within a schedule named **schedule_name** for the server's default **time_zone**. Example: **`timeInValue(%{00094}, count(toStringList(^%, ",")) > 1, "schedule_name", LOCAL)`** returns the number of milliseconds the field components (field code **%{00094}**) of the current issue has contained more than one selected component, within a schedule named **schedule_name** for the server's default **time_zone**. Example: **`timeInValue(%{00017}, ^% in ["Critical", "High"], "schedule_name", LOCAL)`** returns the number of milliseconds the current issue has had a priority value of Critical or High (field code **%{00017}**), within a schedule named **schedule_name** for the server's default **time_zone**.

timeInValue(number field **field**, boolean expression **predicate**, string **schedule_name**, timeZone **time_zone**) Available since version 1.1.0	NUMBER BOOLEAN STRING TIMEZONE	Returns the NUMBER of milliseconds of a number or date-time **field** with code **{nnnnn}** of the current issue has had a values satisfying a boolean expression **predicate**, where the numeric value of the **field** with code **{nnnnn}** is represented by **^**. The time being calculated by this function is only counted during a defined schedule with name **schedule_name** for time zone **time_zone**. Example: **`timeInValue({00012}, ^ != null, "schedule_name", LOCAL)`** returns the number of milliseconds the field due date (field code **{00012}**) of the current issue has had a value, ignoring the case, within a schedule named "my_schedule" for the server's default **time_zone**. Example: **`timeInValue({10001}, ^ >= 5 AND ^ <= 10, "schedule_name", LOCAL)`** returns the number of milliseconds a hypothetical numeric field called Passengers (field code **{10001}**) of the current issue has had a value between 5 and 10, within a schedule named **schedule_name** for the server's default **time_zone**. Example: **`timeInValue({10001}, modulus(^, 2) = 0, "schedule_name", LOCAL)`** returns the number of milliseconds a hypothetical numeric field called Passengers (field code **{10001}**) in current issue has had an even value, within a schedule named **schedule_name** for the server's default **time_zone**.
timeInValue(string field **field**,issue list **issues**, boolean expression **predicate**, string **schedule_name**, timeZone **time_zone**) Available since version 1.1.0	STRING ISSUE [] BOOLEAN STRING TIMEZONE	Returns the NUMBER of milliseconds a string **field** with code **%{nnnnn}** has had a value satisfying a boolean expression **predicate** in distinct issues, where the value of the **field** with code **%{nnnnn}** is represented by **^%**. The time being calculated by this function is only counted during a defined schedule with name **schedule_name** for time zone **time_zone**. Example: **`timeInValue(%{00000}, subtasks(), ^% ~~ "ERROR" OR ^% ~~ "WARNING", "my_schedule", LOCAL)`** returns the sum of milliseconds the fields summary (field code **%{00000}**) of all subtasks of the current issue have have contained any of the words "ERROR" or "WARNING", ignoring the case, within a schedule named "**schedule_name**" for the server's default **time_zone**. Example: **`timeInValue(%{00094}, epic(), count(toStringList(^%, ",")) > 1, "my_schedule", LOCAL)`** returns the number of milliseconds the field components (field code **%{00094}**) in the linked Epic issue has contained more than one selected component, within a schedule named **my_schedule** for the server's default **time_zone**. Example: **`timeInValue(%{00017}, filterByIssueType(linkedIssues(), "Bug, New Feature"), ^% in ["Critical", "High"], "my_schedule", LOCAL)`** returns the sum of milliseconds all linked Bugs and New Features of the current issue have had a priority (field code **%{00017}**) value of Critical or High., within a schedule named **my_schedule** for the server's default **time_zone**.

timeInValue(number field **field**, issue list **issues**, boolean expression **predicate**, string **schedule_name**, timeZone **time_zone**) Available since version 1.1.0	NUMBER ISSUE [] BOOLEAN STRING TIMEZONE	Returns the NUMBER of milliseconds number or date-time **field** with code **{nnnnn}** has had a value satisfying a boolean expression **predicate** in distinct issues, where the numeric value of the **field** with code **{nnnnn}** is represented by **^**. The time being calculated by this function is only counted during a defined schedule with name **schedule_name** for time zone **time_zone**. Example: **`timeInValue({00012}, subtasks(), ^ != null, "schedule_name", LOCAL)`** returns the number of milliseconds the field due date (field code **{00012}**) of all subtasks of the current issue have had a value, within a schedule named "my_schedule" for the server's default **time_zone**. Example: **`timeInValue({10001}, epic(), ^ >= 5 AND ^ <= 10, "schedule_name", LOCAL)`** returns the number of milliseconds a hypothetical numeric field called Passengers (field code **{10001}**) in the linked Epicissue has had a value between 5 and 10, within a schedule named "**schedule_name**" for the server's default **time_zone**. Example: **`timeInValue({10001}, filterByIssueType(linkedIssues(), "Bug, New Feature"), modulus(^, 2) = 0, "schedule_name", LOCAL)`** returns the number of milliseconds a hypothetical numeric field called Passengers (field code **{10001}**) has had an even value in any linked Bug or New Feature, within a schedule named **schedule_name** for the server's default **time_zone**.
fieldChangeTimes(string field **field**, boolean expression **predicate**) Available since version 1.1.0	STRING BOOLEAN	Returns the timestamps as NUMBER [] of when a string value of **field** with code **%{nnnnn}** has changed satisfying a certain **predicate** that depends on the values of the **field** before and after the value change. The string value before the change is represented by **`^0%`**, and after the change by **`^1%`**. The timestamps are returned as a number list sorted in ascending order. Example: **`fieldChangeTimes(%{00000}, ^0% !~~ "IMPORTANT" AND ^1% ~~ "IMPORTANT")`** returns the list of timestamps when word "IMPORTANT" has been added to the current issue's summary (field code **%{00000}**) ignoring the case. Example: **`fieldChangeTimes(%{00017}, ^0% = null AND ^1% != null)`** returns the list of timestamps of when the issue priority (field code **%{00017}**) of the current issue has been set. Example: **`fieldChangeTimes(%{00017}, ^0% not in ["Critical", "High"] AND ^1% in ["Critical", "High"])`** returns the list of timestamps when current issue's priority (field code **%{00017}**) has become Critical or High.

fieldChangeTimes(number field **field**, boolean expression **predicate**) Available since version 1.1.0	NUMBER BOOLEAN	Returns the timestamps as NUMBER [] of when a numeric / date-time value of **field** with code **{nnnnn}** has changed satisfying a certain **predicate** that depends on the values of the field before and after the value change. The numeric value before the change is represented by **`^0`**, and after the change by **`^1`**. The timestamps are returned as a number list sorted in ascending order. Example: **`fieldChangeTimes({00012}, ^0 < ^1)`** returns the timestamps of when the Due date (field code **{00012}**) has been edited to a higher value. Example: **`fieldChangeTimes({10001}, abs(^0 - ^1) / ^0 >= 0.25)`** returns the timestamps of when a hypothetical numeric field called Passengers(field code **{10001}**) has been edited with a variation of at least 25% over its previous value.
fieldChangeTimes(string field **field**, issue list **issues**, boolean expression **predicate**) Available since version 1.1.0	STRING ISSUE [] BOOLEAN	Returns the timestamps as NUMBER [] of when a string value of **field** with code **%{nnnnn}** in distinct parameter issues have changed satisfying certain **predicate** that depends on the values of the fields before and after the value change. The string value before the change is represented by **`^0%`**, and after the change by **`^1%`**. The timestamps are returned as a number list containing a sequence of sorted numeric values in ascending order for each parameter issue. Example: **`fieldChangeTimes(%{00000}, subtasks(), ^0% !~~ "IMPORTANT" AND ^1% ~~ "IMPORTANT")`** returns the list of timestamps of when the word "IMPORTANT" has been added the the summary (field code **%{00000}**) of all current issue's subtasks, ignoring the case. Example: **`fieldChangeTimes(%{00017}, epic(), ^0% = null AND ^1% != null)`** returns the list of timestamps of when the issue **priority** (field code **%{00017}**) of the current issue's epic has been set. Example: **`fieldChangeTimes(%{00017}, linkedIssues("is blocked by"), ^0% not in ["Critical", "High"] AND ^1% in ["Critical", "High"])`** returns the list of timestamps of when the **priority**(field code **%{00017}**) in all blocking linked issues has become Critical or High.
fieldChangeTimes(number field **field**, issue list **issues**, boolean expression **predicate**) Available since version 1.1.0	NUMBER ISSUE [] BOOLEAN	Returns the timestamps as NUMBER [] of when a numeric value of **field** with code **{nnnnn}** in distinct parameter issues have changed satisfying a certain **predicate** that depends on the values of the fields before and after the value change. The numeric value before the change is represented by **`^0`**, and after the change by **`^1`**. The timestamps are returned as a number list containing a sequence of sorted numeric values in ascending order for each parameter issue. Example: **`fieldChangeTimes({00012}, subtasks(), ^0 < ^1)`** returns the timestamps of when the due date (field code **{00012}**) has been edited to a higher value in any of the current issue's subtasks. Example: **`fieldChangeTimes({10001}, epic(), abs(^0 - ^1) / ^0 >= 0.25)`** returns the timestamps when a hypothetical numeric field called Passengers (field code **{10001}**) in the current issue's epic has been edited with a variation of at least 25% over its previous value
lastFieldChangeTime(string field **field**) Available since version 1.1.0	STRING	Returns the timestamp as NUMBER of most recent value update of a **field** with code **%{nnnnn}**. Example: **`lastFieldChangeTime(%{00000})`** returns the timestamp of the last update of an issue's summary (field code **{00000}**).

Time Macros

Date-Time values are numeric values representing the number of **milliseconds** elapsed since **January 1, 1970, 00:00:00 GMT**.

Macros are **aliases for literal / fixed values**. A comprehensive set of time macros is provided to make your expressions more readable.

Macro	Equivalent value
`{SECOND}`	`1000`
`{MINUTE}`	`1000 * 60`
`{HOUR}`	`1000 * 60 * 60`
`{DAY}`	`1000 * 60 * 60 * 24`
`{WEEK}`	`1000 * 60 * 60 * 24 * 7`
`{MONTH}`	`1000 * 60 * 60 * 24 * 30`
`{YEAR}`	`1000 * 60 * 60 * 24 * 365`

The following macros are available to be used with function **`dayOfTheWeek(t, time_zone)`**:

Macro	Equivalent value
`{SUNDAY}`	`1`
`{MONDAY}`	`2`
`{TUESDAY}`	`3`
`{WEDNESDAY}`	`4`
`{THURSDAY}`	`5`
`{FRIDAY}`	`6`
`{SATURDAY}`	`7`

The following macros are available to be used with function **`month(t, time_zone)`**:

Macro	Equivalent value
`{JANUARY}`	`1`
`{FEBRUARY}`	`2`
`{MARCH}`	`3`
`{APRIL}`	`4`
`{MAY}`	`5`
`{JUNE}`	`6`
`{JULY}`	`7`
`{AUGUST}`	`8`
`{SEPTEMBER}`	`9`
`{OCTOBER}`	`10`
`{NOVEMBER}`	`11`
`{DECEMBER}`	`12`

Examples

Input	Output
`(2 * 6) / 3`	Returns the result of a **simple calculation: 4**
`{...duedate} + 2 * {DAY}`	Returns a **date** which is **two days in the future** of the current **Due Date**.
`round(({...duedate} - {...currentDateTime}) / {HOUR})`	Returns the number of hours from between the **current date and time** to **Due Date**.

Strings

Fixed values

- Texts or strings need to be written in **double quotes**, e.g., "This is a string literal."
- Operator + is used for **concatenating string**. e.g., "This is" + " a string." = "This is a string." .
- The **Escape character** is "\" . This character can precede any of the following characters: **", \, n, r, t, f** and **b** in order to invoke an alternative interpretation.
  For example, if you want to introduce a double quote in a string literal you should precede it with escape character \ as in "The man said: `\"Hello!\"."` , where we are using escape character \ to write string Hello! in double quotes.

Variable values (field values)

Text / String field values can be inserted in expressions using field codes with format **%{...somefield}**, or **%{...somefield.i}** for referencing concrete levels in cascading select fields (i = 0 for base level).

Pro tip

For checking if a field has a value you can use `%{...somefield} = null` or `%{...somefield} != null`.
For a concrete level in a **Cascading Select** or **Multi-Cascading Select** field, you should use `%{...somefield.i} = null` or `%{...somefield.i} != null`.

Any field type has a string value, so you can also use **%{...somefield}** to insert string values of fields of types: **Number, Date, Date-Time** and **Priority**.

String Functions

Function	Input	Returned value
trim(string **s**)	STRING	Returns a copy of the STRING without leading and trailing blanks (space and tab characters). Example: `trim(" Hello World! ")` returns **"Hello World!"**.
substring(string **s**, number **beginIndex**, number **endIndex**)	STRING	Returns a substring of the STRING beginning at index **beginIndex** and ending at **endIndex - 1**. Thus the length of the substring is **endIndex-beginIndex**. Example: `substring("smiles", 1, 5)` returns "**mile**".
toUpperCase(string **s**)	STRING	Returns STRING with all its characters converted to upper case. Example: `toUpperCase("heLLo WORLD!")` returns "**HELLO WORLD!**".
toLowerCase(string **s**)	STRING	Returns STRING **s** with all its characters converted to lower case. Example: `toLowerCase("heLLo WORLD!")` returns "**hello world!**".

capitalizeWords(string **s**)	STRING	Capitalizes all the whitespace separated words in STRING. Example: **`capitalizeWords("heLLo WORLD!")`** returns "**HeLLo WORLD!**".
capitalizeWordsFully(string **s**)	STRING	Converts all the whitespace separated words in STRING into capitalized words, that is each word is made up of a titlecase character and then a series of lowercase characters. Example: **`capitalizeWordsFully("heLLo WORLD!")`** returns "**Hello World!**".
replaceAll(string **s**, string **regexp**, string **replacement**)	STRING REGEX	Returns a copy of **s** where each substring matching the given **regular expression** **regexp** has been replaced with the given **replacement** string. Example: **`replaceAll(" Hello World ", "\\s", "")`** returns "**HelloWorld**" .
replaceFirst(string **s**, string **regexp**, string **replacement**)	STRING REGEX	Returns a copy of STRING where the first substring matching the given **regular expression** **regexp** has been replaced with the given **replacement** string. Example: **`replaceFirst("Hello World", "l", "_")`** returns "**He_lo World**" .
matches(string **s**, string **regexp**)	STRING REGEX	Returns a BOOLEAN value **true** if string s matches **regular expression** **regexp**, otherwise returns **false**. Example: **`matches("readme.txt", ".*\\.txt$")`** returns **true** .
findPattern(string **s**, string **regexp**)	STRING REGEX	Returns a STRING [] with all substrings in argument **s** matching **regular expression** in string argument **regexp**. Example: **`findPattern("Between 1900 and 2000 world population increase from 1.5 to 6.1 billions.", "\\d+(\\.\\d+)?")`** returns **["1900", "2000", "1.5", "6.1"]** .
findPatternIgnoreCase(string **s**, string **regexp**)	STRING [] REGEX	Returns a STRING [] with all substrings in argument **s** matching **regular expression** in string argument **regexp**. Evaluation of the regular expression is carried out in ignoring case mode. Example: **`findPatternIgnoreCase("Grass is Green and Sky is Blue.", "red|green|blue")`** returns **["Green", "Blue"]** .
findModify(string **s**, string **regexp**, string **replacement_expression**)	STRING REGEX	Returns a STRING like **s**, but where all substrings matching **regexp** have been replaced with the result of evaluating **replacement_expression** against each these substrings. Argument **text_expression** is an expression that returns a **string**, where **^%** represents each of the matching substrings, and **^** represents the order of appearance beginning with **1**. Example: **`findModify("The cure for boredom is curiosity.", "[aeiou]", modulus(^, 2) = 1 ? toUpperCase(^%) : ^%)`** returns **"ThE curE for bOredOm is cUriOsity."** .
findReplaceAll(string **s**, string **find**, string **replacement**)	STRING	Returns a STRING with content of argument **s** where every ocurrence of substring find has been replaced with string **replacement**. Example: **`findReplaceAll("Goodbye my love, hello my friend.", "my", "your")`** returns "**Goodbye your love, hello your friend.**" .
findReplaceAllIgnoreCase(string **s**, string **find**, string **replacement**)	STRING	Returns a STRING with content of argument **s** where every ocurrence of substring **find**, ignoring the case, has been replaced with string **replacement**. Example: **`findReplaceAllIgnoreCase("Hello my love, hello my friend.", "hello", "Goodbye")`** returns "**Goodbye my love, Goodbye my friend.**" .

findReplaceFirst(string **s**, string **find**, string **replacement**)	STRING	Returns a STRING with content of argument **s** where first ocurrence of substring **find** has been replaced with string **replacement**. Example: `findReplaceFirst("Goodbye my love, hello my friend.", "my", "your")` returns "**Goodbye your love, hello my friend.**" .
findReplaceFirstIgnoreCase(string **s**, string **find**, string **replacement**)	STRING	Returns a STRING with content of argument **s** where first ocurrence of substring **find**, ignoring the case, has been replaced with string **replacement**. Example: `findReplaceFirstIgnoreCase("Goodbye my love, hello my friend.", "My", "your")` returns "**Goodbye your love, hello my friend.**" .
length(string **s**)	STRING	Returns a NUMBER with the length of **s**. Example: `length("Star Wars")` returns **9** .
getAscii(number **code**)	NUMBER	Returns a STRING containing the symbol corresponding to a extended ASCII **code** (0 <= **code** <= 255). Example: `getAscii(65)` returns "**A**" .
similarity(string **s1**, string **s2**)	STRING	Returns a NUMBER value between 0 and 100 representing the percentage of similarity between two strings based on the Jaro Winkler similarity algorithm . **100** represents **full equivalence**, and **0** represents **zero similarity** between both string arguments. Examples: `similarity("Automation Toolbox for Jira", "Automation Toolbox for Jira")` returns **100** `similarity("Automation Toolbox for Jira", "Jira WorflowTolbox")` returns **97** `similarity("My Gym. Childrens Fitness", "My Gym Children's Fitness Center")` returns **92** `similarity("D N H Enterprises Inc", "D & H Enterprises, Inc.")` returns **91** `similarity("ABC Corporation", "ABC Corp'")` returns **92** `similarity("Hello World!", "Bye bye World!")` returns **69** `similarity("I caught a lizard", "This is my giraffe")` returns **51**
escapeHTML(string **s**)	STRING	Escapes the characters in a STRING using HTML entities. Example: `escapeHTML("<Français>")` returns "**< Français>**" .
unescapeHTML(string **s**)	STRING	Unescapes STRING containing entity escapes to a string containing the actual Unicode characters corresponding to the escapes. Example: `unescapeHTML(""bread" & "butter"")` returns "**\"bread\" & \"butter\"**" .
wikiToHTML(string **s**)	STRING	Renders rich text wiki content of STRING into HTML. Example: `wikiToHTML("+Hello *world*!+")` return "** <ins>Hello world!</ins> **" .
htmlToTxt(string **s**)	STRING	Renders HTML content of STRING into plain text by removing all the html tags. Example: `wikiToHTML("<p>Hello <b>world</b>!</p>")`return "**Hello world!**" .

Examples

Input	Output
`"Hello" + " " + "world" + "."`	**Hello world.**
`trim(%{...summary})`	**Summary** of an issue **without leading** and **trailing bla nks**
`%{...description} + "\nLAST USER: " + toUpperCase(%{...currentUser})`	**Description** of an issue and a new line with string "**LA ST USER:** " and the **name of current user** in upper case.

Issue lists

Overview

The **Issue list** data type is an ordered list of issues.

This data type is returned by functions returning selections of issues (**linked issues, sub-tasks, issues in a project**, or subsets).

Example

An issue list with 5 elements: **`[HR-1,HR-2,HR-3]`**

Issue list functions

Issue list functions either return **issue lists** (e.g. [issuekey-1,issuekey-2,issuekey3,...]) or **strin g lists** or **number lists** for retrieving **issue fields**

The following functions are intended to build expressions that reference **linked issues**, **sub-tasks**, or doing any kind of **issue selection**, and for retrieving their field values.

Function	Input	Returned value
subtasks()		Returns the ISSUE [] of sub-tasks of current issue.
subtasks(iss ue list **issues**)	ISSUE []	Returns the ISSUE [] of sub-tasks of issues in argument **issues**. Duplicated issues in argument **issues** are discarded. Example: **`subtasks(linkedIssues())`** returns the list of sub-tasks of linked issues.
subtasks(str ing **issue_ke ys**)	STRING	Returns the ISSUE [] of sub-tasks of issues whose keys are in **issue_keys**. Argument **issue_keys** is a comma separated list of issue keys. Duplicated issue keys in argument **i ssue_keys** are discarded. Example: **`subtasks(%{...parentIssuekey})`** returns the list of sub-tasks of parent issue, i.e., sibling sub-tasks plus current sub-task.
siblingSubt asks()		Returns the ISSUE [] of sibling sub-tasks of current issue, i.e., all sub-tasks with the same parent as current issue, except current issue. In case current issue is not a sub-task, an empty issue list will be returned. Note that **`siblingSubtasks()`** is equivalent to **`subtasks(%{...parentIssuekey}) EXCEPT issueKeysToIssueList(%{...Issuekey})`** , where **%{...parentIssuekey}** is Parent's issue key and **%{... Issuekey}** is Issue key.
siblingSubt asks(issue list **issues**)	ISSUE []	Returns the ISSUE [] of sibling sub-tasks of issues in argument **issues**, provided they are sub-tasks. Duplicated issues in argument **issues** are discarded.

siblingSubtasks(string **issue_keys**)	STRING	Returns the ISSUE [] of sibling sub-tasks of issues whose keys are in **issue_keys**, provided they are sub-tasks. Argument **issue_keys** is a comma separated list of issue keys. Duplicated issue keys in argument **issue_keys** are discarded.
linkedIssues()		Returns the ISSUE [] of issues linked to current issue, including **Epic-Task** links. An issue appears in the output as many times as is linked to current issue. Function **distinct**(issue list) can be used to remove duplicated issues. Example: `distinct(linkedIssues() EXCEPT linkedIssues("has Epic, is Epic of"))` returns all the issues linked to current issue, excluding **Epic-Task** issue links.
linkedIssues(string **issue_link_types**)	STRING	Returns the ISSUE [] of issues linked to current one using issue link types in argument **issue_link_types**. Argument **issue_link_types** is a comma separated list of issue link type names, or an empty string ("") for representing all issue link types, i.e., `linkedIssues("")` is equivalent to `linkedIssues()` . Example: `linkedIssues("blocks, clones")` returns all issues linked with to current issue using issue link types blocks or clones.
linkedIssues(string **issue_link_types**, issue list **issues**)	STRING ISSUE []	Returns the ISSUE [] of issues linked to those ones in argument **issues** using issue link types in argument **issue_link_types**. Duplicated issues in argument **issues** are discarded. Example: `linkedIssues("", subtasks())` returns all issues linked to current issue's sub-tasks using any issue link type.
linkedIssues(string **issue_link_types**, string **issue_keys**)	STRING	Returns the ISSUE [] of issues linked to those ones whose keys are in argument **issue_keys**. Argument **issue_keys** is a comma separated list of issue keys. Duplicated issue keys in argument **issue_keys** are discarded. Example: `linkedIssues("is blocked by", %{... parentIssuekey})` returns all issues blocking parent issue.
transitionLinkedIssues(string **issue_link_types**)	STRING	Returns the ISSUE [] of issues linked to current one with links created in current transition screen using issue link types in argument **issue_link_types**. Argument **issue_link_types** is a comma separated list of issue link type names, or an empty string `("")` for representing all issue link types, i.e., `transitionLinkedIssues("")` is equivalent to `transitionLinkedIssues()` . This function is useful for validating only new issue links created by user in transition screen. Example: `transitionLinkedIssues("blocks, clones")` returns the list of issues linked in current transition's screen using issue link types **blocks** and **clones**.
transitivelyLinkedIssues(string **issue_link_types**)	STRING	Returns the ISSUE [] of issues directly or transitively linked to current issue using issue link types in argument **issue_link_types**. Argument **issue_link_types** is a comma separated list of issue link type names, or an empty string `("")` for representing all issue link types. Example of transitive link: if **ISSUE-1 blocks ISSUE-2 blocks ISSUE 3**, then **ISSUE-1** is blocking transitively **ISSUE-3**.
transitivelyLinkedIssues(string **issue_link_types**, issue list **issues**)	STRING ISSUE []	Returns the ISSUE [] of issues directly or transitively linked to those ones in argument **issues** using issue link types in argument **issue_link_types**. Argument **issue_link_types** is a comma separated list of issue link type names, or an empty string `("")` for representing all issue link types.
transitivelyLinkedIssues(string **issue_link_types**, string **issue_keys**)	STRING	Returns the ISSUE [] of issues directly or transitively linked to those ones in argument **issue_keys** using issue link types in argument **issue_link_types**. Argument **issue_link_types** is a comma separated list of issue link type names, or an empty string `("")` for representing all issue link types.

epic()		Returns an ISSUE [] containing current issue's epic, in case current issue is directly under an epic (e.g., a **Story**). If current issue is a sub-task, then the epic of its parent issue is returned. If current issue is an epic itself, then current issue is returned.
epic(issue list **issues**)	ISSUE []	Returns the ISSUE [] of epic issues under which those issues in argument **issues** are. If some of those issues are sub-tasks, then the epic of their parent is returned. Duplicated issues in argument issues are discarded. Output can contain duplicated issues. Example: **`epic(linkedIssues("is blocked by"))`** returns the list of epics of those issues which are blocking current issue.
epic(string **issue_keys**)	STRING	Returns the ISSUE [] of epic issues under which those issues with keys in **issue_keys** are. If some of those issues are sub-tasks, the epic of their parent is returned. Argument **issue_keys** is a comma separated list of issue keys. Duplicated issue keys in argument **issue_keys** are discarded. Output can contain duplicated issues. Example: **`epic("CRM-15, HD-21")`** returns the list of epics under which issues with keys **CRM-15** and **HD-21** are.
issuesUnderEpic()		Returns an ISSUE [] containing issues which are directly under current issue's epic (i.e., **Stories** are included in the output, but their **sub-tasks** are not). Current issue's epic is obtained using the logic of function epic() . Current issue is included in the output, except if current issue is an epic itself.
issuesUnderEpic(issue list **issues**)	ISSUE []	Returns an ISSUE [] containing issues which are directly under the epic of issues in argument **issues**. Duplicated issues are filtered from output. Example: **`issuesUnderEpic(linkedIssues("is blocked by"))`** returns the list of issues directly under epics of issues blocking current issue.
issuesUnderEpic(string **issue_keys**)	STRING	Returns an ISSUE [] containing issues which are directly under the epic of issues with keys in argument **issue_keys**. Argument **issue_keys** is a comma separated list of issue keys. Duplicated issues are filtered from output. Example: **`issuesUnderEpic("CRM-15, HD-21")`** returns the list of issues directly under epic of issues with keys **CRM-15** and **HD-21**.
siblingIssuesUnderEpic()		Returns an ISSUE [] containing issues which are directly under epic of current issue (i.e., **Stories** are included in the output, but their **sub-tasks** are not), excluding current issue. Current issue should be an issue directly under an epic, (i.e., it can't be a **sub-task** or an **epic**).
siblingIssuesUnderEpic(issue list **issues**)	ISSUE []	Returns an ISSUE [] containing issues which are directly under the epic of issues in argument **issues**, excluding issues in argument **issues** from the output. Duplicated issues are filtered from output. Example: **`siblingIssuesUnderEpic(linkedIssues("is blocked by"))`** returns the list of issues directly under epics of issues blocking current issue, excluding from the output issues blocking current issue.
siblingIssuesUnderEpic(string **issue_keys**)	STRING	Returns an ISSUE [] containing issues which are directly under the epic of issues with keys in argument **issue_keys**, excluding from the output issues with keys in argument **issue_keys**. Argument **issue_keys** is a comma separated list of issue keys. Duplicated issues are filtered from output. Example: **`siblingIssuesUnderEpic("CRM-15, HD-21")`** returns the list of issues directly under epic of issues with keys **CRM-15** and **HD-21**, excluding from the output issues with keys **CRM-15** and **HD-21**.

issuesFromJQL(string **jql_query**)	STRING	Returns the ISSUE [] resulting of the execution of a JQL query represented by string argument **jql_query**. Visibility permissions applied are those of **current user**. We advice to use this function for performance reasons when the number of issues to be retrieved or filtered is very high (all issues in a project or various projects). Typically you will want to use this function for replacing any current expression using **getIssuesFromProjects()** function.
issuesFromJQL(string **jql_query**, string **user_name**)	STRING	Returns the ISSUE [] resulting of the execution of a JQL query represented by string argument **jql_query**. Visibility permissions applied are those of user in argument **user_name**. We advice to use this function for performance reasons when the number of issues to be retrieved or filtered is very high (all issues in a project or various projects). Typically you will want to use this function for replacing any current expression using **getIssuesFromProjects()** function.
filterByIssueType(issue list **issues**, string **issue_types**)	ISSUE [] STRING	Filters ISSUE [] in argument **issues**, leaving only those issue types appearing in argument **issue_types**. Argument **issue_types** is a comma separated list of issue type names. Example: **`filterByIssueType(subtasks(), "Bug, Improvement, New Feature")`** returns the list of sub-tasks with issue types **Bug**, **Improvement** or **New Feature**.
filterByStatus(issue list **issues**, string **statuses**)	ISSUE [] STRING	Filters ISSUE [] in argument **issues**, leaving only those ones in statuses appearing in argument **statuses**. Argument **statuses** is a comma separated list of status names. Example: **filterByStatus(linkedIssues("is blocked by"), "Open, Reopened, In Progress")** returns the list of blocking issues in statuses **Open**, **Reopened** or **In Progress**.
filterByStatusCategory(issue list **issues**, string **status_categories**)	ISSUE [] STRING	Filters ISSUE [] in argument **issues**, leaving only those ones in statuses with categories in **status_categories**. Argument **status_categories** is a comma separated list of status category names. Example: **`filterByStatusCategory(linkedIssues("is blocked by"), "New, In Progress")`** returns the list of blocking issues in statuses with categories **New** or **In Progress**.
filterByResolution(issue list **issues**, string **resolutions**)	ISSUE [] STRING	Filters ISSUE [] in argument **issues**, leaving only those ones with resolutions appearing in argument **resolutions**. Argument **resolutions** is a comma separated list of resolution names. If this argument receives an empty string **`("")`**, the function will return issues with unset field Resolution. Example: **`filterByResolution(subtasks(), "Won't Fix, Cancelled")`** returns the list of sub-tasks with resolutions **Won't Fix** or **Cancelled**.
filterByProject(issue list **issues**, string **projects**)	ISSUE [] STRING	Filters ISSUE [] in argument **issues**, leaving only those ones in projects present at argument **projects**. Argument **projects** is a comma separated list of project keys. Example: **`filterByProject(linkedIssues(), "CRM, HR")`** returns the list of linked issues belonging to projects with keys **CRM** or **HR**.
filterByProjectCategory(issue list **issues**, string **project_categories**)	ISSUE [] STRING	Filters ISSUE [] in argument **issues**, leaving only those ones in projects with category in **project_categories**. Argument **project_categories** is a comma separated list of project category names. Example: **`filterByProjectCategory(linkedIssues(), "Development, Production")`** returns the list of linked issues belonging to projects in categories keys **Development** or **Production**.
filterByFieldValue(issue list **issues**, numeric field **field**, comparison operator **operator**, number **n**)	ISSUE [] NUMBER	Filters ISSUE [] in argument **issues**, leaving only those issues where logical predicate formed by arguments **field operator n** is evaluated as true. Available comparison operators are **`=, !=, <, <=, >`** and **>=** . Argument **field** has format **{...somefield}**. Example: **`filterByFieldValue(subtasks(), {00079}, >, 1)`** returns sub-tasks with more than one **Affects Version/s**.

filterByFieldValue(issue list **issues**, string field **field**, comparison operator **operator**, string **s**)	ISSUE [] STRING	Filters ISSUE [] in argument **issues**, leaving only those issues where logical predicate formed by arguments **field operator s** is evaluated as true. Available comparison operators are **`=, !=, <, <=, >, >=, ~, !~, in`** and **`not in`**. Case ignoring operators are also available: **`=~, !=~, ~~, !~~, in~`** and **`not in~`**. Argument **field** has format **%{...somefield}** for string fields, or **%{...somefield.i}** for cascading select fields. Example: **`filterByFieldValue(linkedIssues(), %{...components}, ~, "Web")`** returns linked issues with component "**Web**".
filterByCardinality(issue list **l**, comparison operator **operator**, number **n**)	ISSUE [] NUMBER	Returns ISSUE [] in **l** whose cardinality (i.e., the number of times it appears in list **l**) satisfies the comparison cardinality **operator n**. Available comparison operators: **`=, !=, <, <=, >`** and **`>=`** . Example: **`filterByCardinality(linkedIssues(), >, 1)`** returns a list with all issues linked to current issue with 2 or more issue links.
append(issue list **l**, issue list **m**)	ISSUE []	Returns ISSUE [] with all issues in arguments **l** and **m**. Duplicated issues may appear in output. Use function **union(l, m)** instead, if you want to avoid repetitions. Example: **`append(linkedIssues("is blocked by"), subtasks())`** returns the list blocking issues plus sub-tasks. If a sub-task is also linked with issue link type "**is blocked by**", it will appear twice in the output list.
union(issue list **l**, issue list **m**)	ISSUE []	Returns ISSUE [] with all issues in argument **l** or in argument **m** without duplicated issues. Example: **`union(linkedIssues(), subtasks())`** returns the list of linked issues and sub-tasks of current issue, without issue repetitions.
except(issue list **l**, issue list **m**)	ISSUE []	Returns ISSUE [] with all issues in argument **l** which are not in argument **m**. Duplicated issues in **l** may appear in output. Use function **distinct()** to remove them if you need to. Example: **`except(linkedIssues(), subtasks())`** returns the list of linked issues removing those which are also sub-tasks of current issue.
intersect(issue list **l**, issue list **m**)	ISSUE []	Returns ISSUE [] with all issues in argument **l** and **m** simultaneously. Example: **`intersect(linkedIssues(), subtasks())`** returns the list of linked issues which are also sub-tasks of current issue.
distinct(issue list **l**)	ISSUE []	Returns ISSUE [] with all issues in list **l** without any duplication. Example: **`distinct(linkedIssues())`** returns the list of linked issues, with only one occurrence per issue, although an issue may be linked with more than one issue link type.
fieldValue(string field **field**, issue list **issues**)	STRING ISSUE []	Returns the STRING [] of string values stored in argument **field** in those issues in argument **issues**. Argument field has format **%{...somefield}**, or **%{...somefield.i}** for cascading select fields. The number of values in output is the number of issues in argument **issues** with **field** set, except for multi-valued fields, for which a value is returned for each selected value in the field. Multi-valued fields are fields of types **Multi Select, Checkboxes, Components, Versions, Multi User Picker, Multi Group Picker, Issue Pickers, Attachments** and **Labels**. Example: **`fieldValue(%{...reporter}, subtasks())`** returns the list of **reporter users** of sub-tasks.
fieldValue(numeric field **field**, issue list **issues**)	NUMBER ISSUE []	Returns the NUMBER [] of numeric values stored in argument **field** in those issues in argument **issues**. Argument **field** has format **{...somefield}**. The number of values in output is the number of issues in argument **issues** with **field** set. Example: **`fieldValue({...duedate}, subtasks())`** returns the list of **Due Dates** of sub-tasks.

textOnIssueList(issue list **issues**, string **text_expression**)	ISSUE [] STRING	Returns a STRING [] resulting of evaluating **text_expression** against each of the issues in argument **issues**. Argument **text_expression** is an expression that returns a **string**, where references to field values of issues in argument issues are done with prefix **^** before field code, e.g., **^%{...summary}** is field code for Summary in each of the issues in argument **issues**. Example: `textOnIssueList(subtasks(), ^%{...assignee} = ^%{...reporter} ? ^%{...Issuekey} : null)` returns the issue keys of sub-tasks with same user as reporter and as assignee.
mathOnIssueList(issue list **issues**, number **math_time_expression**)	ISSUE [] NUMBER	Returns a NUMBER [] resulting of evaluating **math_time_expression** against each of the issues in argument **issues**. Argument **math_time_expression** is a math/time expression, where references to field values of issues in argument issues are done with prefix **^** before field code, e.g., **^{...duedate}** is field code for **Due date** in each of the issues in argument **issues** . Example: `mathOnIssueList(linkedIssues("is blocked by"), (^{...duedate} != null ? ^{...duedate} - ^{...created} : 0) / {HOUR})` returns a list of numbers with the number of days from issue creation to due date for all issues linked using "**is blocked by**" issue link type.
numberOfRemoteIssueLinks(string **issue_link_types**)	STRING	Returns the NUMBER of issue links to other Jira instances using any of the issue link types in argument **issue_link_types**. Argument **issue_link_types** is a comma separated list of issue link type names, or empty string `("")` for representing all issue link types.
count(issue list **l**)	ISSUE []	Returns the NUMBER of issues in **l**. Example: `count(filterByResolution(linkedIssues("is blocked by"), ""))` returns the number of non-resolved blocking issues.
getIssuesFromProjects(string **projects**)	STRING	Returns an ISSUE [] with all issues of projects in argument **projects**. Argument **projects** is a string containing a comma separated list of project keys or project names. Example: `getIssuesFromProjects("CRM, HT")` returns all issues in project **CRM** and **HT**. This function can make your expression run slowly due to the high number of issues retrieved and needing to be filtered. Using **issuesFromJQL()** for retrieving and filtering issues will make your expression run much faster.
first(issue list **l**)	ISSUE []	Returns an ISSUE [] with the first element in issue list **l**, or an empty list if **l** is an empty list.
last(issue list **l**)	ISSUE []	Returns an ISSUE [] with the last element in issue list **l** , or an empty list if **l** is an empty list.
nthElement(issue list **l**, number **n**)	ISSUE [] NUMBER	Returns an ISSUE [] with the element at position **n** in issue list **l**, where **n >= 1** and **n <= count(l)**. Returns an **empty list** if **n** is greater than the number of elements in **l**.
sublist(issue list **l**, number **indexFrom**, number **indexTo**)	NUMBER	Returns an ISSUE [] with elements in **l** from **indexFrom** index to **indexTo** index. Having **indexFrom >= 1** and **indexFrom <= count(l)** and **indexTo >= 1** and **indexTo <= count(l)** and **indexFrom <= indexTo**.
indexOf(string **issue_key** , issue list **l**)	STRING	Returns the index NUMBER in issue list **l** of issue with key **issue_key**. **Zero** is returned when issue is not found in **l**.
indexOf(issue list **element** , issue list **l**)	ISSUE []	Returns the index NUMBER in issue list **l** of first issue in **element**. **Zero** is returned when first issue in **element** is not found in **l**.

sort(issue list **l**, field **field**, **order**)	ISSUE []	Returns an ISSUE [] with elements in **l** ordered according to values of **field**. Argument **field** has format **{...somefield}** for numeric and date-time fields, **%{...somefield}** for string fields, or **%{...somefield.i}** for cascading select fields. Available orders are **ASC** (for ascending order) and **DESC** (for descending order). Example: **`sort(linkedIssues("is blocked by"), {...duedate}, ASC)`** returns the list of issues blocking current issue, sorted in ascending order by **Due date**.

Examples

Input	Output
`subtasks()`	Returns the list of sub-tasks of the current issue.
`linkedIssues("is blocked by, is caused by")`	Returns the list of issues linked to current one through issue link types "**is blocked by**" and "**is caused by**".
`filterByIssueType(linkedIssues(), "Bug, Incident")`	Returns the list of linked issues with issue type "**Bug**" or "**Incident**".
`filterByPredicate(siblingSubtasks(), %{...resolution} != null)`	Returns the list of sibling sub-tasks (i.e., sub-tasks of same parent as current sub-task) which are not resolved.

Number lists

Overview

The **Number list** data type is an ordered list of numbers. This data type is returned, among others, by functions that return values of number fields in a selection of issues (**linked issues**, **sub-tasks**, and **subsets**).

Fixed values

A **number list** can also be written in literal form using the following format: ***[number, number, ...]***.

> **Example**
>
> A number list with 5 elements: **`[1, -2, 3, 3.14, 2.71]`**

Number list functions

The following functions are intended to build expressions that return **number lists** or **numbers**.

Function	Input	Returned value
filterByCardinality(number list **l**, comparison operator **operator**, number **n**)	NUMBER [] NUMBER	Returns a NUMBER [] **l** whose cardinality (i.e., the number of times it appears in list **l**) satisfies the comparison cardinality **operator n**. Available comparison operators: **`=, !=, <, <=, >`** and **>=**. Example: **`filterByCardinality([1, 1, 2, 3, 4, 4, 4, 5], >, 1)`** returns the following number list: **[1, 4]**.
filterByValue(number list **l**, comparison operator **operator**, number **n**)	NUMBER [] NUMBER	Returns a NUMBER [] **l** satisfying the comparison number_in_list **operator n**. Example: **`filterByValue([1, 2, 3, 10, 11, 25, 100], >, 10)`** returns the list of numbers greater than **10**. i.e., **[11, 25, 100]**

filterByPredicate(number list **l**, boolean expression **predicate**)	NUMBER [] BOOLEAN	Returns a NUMBER [] **l** that validates a predicate. Argument **predicate** is a boolean expression, where **^** is used for referencing numeric values in argument **l**. Example: **`filterByPredicate([1, 2, 3, 4], ^ > 2)`** returns values greater than **2**, i.e., **[3, 4]** . Example: **`filterByPredicate([1, 2, 3, 4], remainder(^, 2) = 0)`** returns even values, i.e., **[2, 4]** .
append(number list **l**, number list **m**)	NUMBER []	Returns a NUMBER [] with all numbers in arguments **l** and **m**. Duplicated numbers may appear in output. Use function **union(l, m)** instead, if you want to avoid repetitions. Example: **`append([1, 2, 3], [3, 4, 5])`** returns **[1, 2, 3, 3, 4, 5]** . Example: **`append(fieldValue({00025}, linkedIssues("is blocked by")), fieldValue({00025}, subtasks()))`** returns a list of numbers with **Total Time Spent (in minutes)** in blocking issues and sub-tasks. This number list can be summed using function **sum()**.
union(number list **l** , number list **m**)	NUMBER []	Returns a NUMBER [] with all numbers in argument **l** or in argument **m** without duplicated numbers. Example: **`union([1, 2, 3], [3, 4, 5])`** returns **[1, 2, 3, 4, 5]** .
except(number list **l**, number list **m**)	NUMBER []	Returns a NUMBER [] with all numbers in argument **l** which are not in argument **m**. Duplicated numbers in l may appear in output. Use function **distinct()** to remove them if you need to. Example: **`except([1, 2, 3, 4, 5], [2, 4])`** returns **[1, 3, 5]**.
intersect(number list **l**, number list **m**)	NUMBER []	Returns a NUMBER [] with all numbers in argument **l** and **m** simultaneously. Example: **`intersect([1, 2, 3, 4, 5], [9, 7, 5, 3, 1])`** returns **[1, 3, 5]** .
distinct(number list **l**)	NUMBER []	Returns a NUMBER [] with all numbers in list **l** without any duplication. Example: **`distinct([1, 2, 1, 3, 4, 4, 5])`** returns **[1, 2, 3, 4, 5]** . Example: **`distinct(fieldValue({...duedate}, linkedIssues("is cloned by")))`** returns a list of dates containing due dates of cloning issues, with only one occurrence per due date, although more than one issue may share the same due date.
count(number list **l**)	NUMBER []	Returns the NUMBER of numeric values in **l**. Example: **`count([1, 1, 2, 2])`** returns **4** . Example: **`count(subtasks()) - count(fieldValue({...duedate}, subtasks()))`** returns the number of sub-tasks with field "**Due Date**" unset.
count(number **n**, number list **l**)	NUMBER NUMBER []	Returns the NUMBER of times **n** appears in **l**. Example: **`count(1, [1, 1, 2, 2, 1, 0])`** returns **3** .
sum(number list **l**)	NUMBER []	Returns the sum of NUMBER values in **l**. Example: **`sum([1, 2, 3, 4, 5])`** returns **15**. Example: **`sum(fieldValue({00025}, subtasks()))`** returns the total time spent in minutes in all sub-tasks of current issue.
avg(number list **l**)	NUMBER []	Returns the arithmetic mean of NUMBER values in **l**. Example: **`avg([1, 2, 3, 4, 5])`** returns **3**. Example: **`avg(fieldValue({00024}, linkedIssues("is blocked by")))`** returns the mean of remaining times in minutes among blocking issues.

max(number list **l**)	NUMBER []	Returns the maximum NUMBER value in **l**. Example: **`max([1, 2, 5, 4, 3])`** returns **5**. Example: **`max(fieldValue({00024}, linkedIssues ("is blocked by")))`** returns the maximum remaining times in minutes among blocking issues.
min(number list **l**)	NUMBER []	Returns the minimum NUMBER value in **l**. Example: **`min([2, 1, 5, 4, 3])`** returns **1**. Example: **`min(fieldValue({00024}, linkedIssues ("is blocked by")))`** returns the minimum remaining times in minutes among blocking issues.
first(number list **l**)	NUMBER []	Returns NUMBER of the first element in number list **l**, or null if **l** is an empty list. Example: **`first([3, 2, 1, 0])`** returns **3**.
last(number list **l**)	NUMBER []	Returns NUMBER of the first element in number list **l**, or null if **l** is an empty list. Example: **`last([3, 2, 1, 0])`** returns **0**.
nthElement(number list **l**, number **n**)	NUMBER [] NUMBER	Returns NUMBER element at position **n** in number list **l**, where **n >= 1** and **n <= count(l)**. Returns null if n is greater than the number of elements in **l**. Example: **`nthElement([5, 6, 7, 8], 3)`** returns **7**.
getMatchingValue(string **key**, string list **key_list**, number list **value_list**)	STRING NUMBER [] STRING []	Returns NUMBER in **value_list** that is in the same position as string key is in **key_list**, or in case key doesn't exist in **key_list** and **value_list** has more elements than **key_list**, the element of **value_list** in position **`count(key_list) + 1`**. Example: **`getMatchingValue("Three", ["One", "Two", "Three", "Four", "Five"], [1, 1+1, 3*1, 4, 4+1])`** returns **3**.
getMatchingValue(string **key**, string list **key_list**, number list **value_list**)	NUMBER [] STRING []	Returns NUMBER value in **value_list** that is in the same position as numeric key is in **key_list**, or in case key doesn't exist in **key_list** and **value_list** has more elements than **key_list**, the element of **value_list** in position **`count(key_list) + 1`**. Example: **`getMatchingValue(5, [1, 3, 5, 7, 9], [1, 1+1, 3*1, 4, 4+1])`** returns **3**.
sublist(number list **l**, number **indexFrom**, number **indexTo**)	NUMBER []	Returns a NUMBER [] with elements in **l** from **indexFrom** index to **indexTo** index. Having **indexFrom >= 1** and **indexFrom <= count(l)** and **indexTo >= 1** and **indexTo <= count(l)** and **indexFrom <= indexTo**. Example: **`sublist([1, 2, 3, 4, 5], 2, 4)`** returns **[2, 3, 4]**.
indexOf(number **element**, number list **l**)	NUMBER NUMBER []	Returns the index of NUMBER value element in number list **l**. Zero is returned when element is not found in **l**. Example: **`indexOf(1, [5, 2, 1, 4, 1])`** returns **3**.
sort(number list **l**, **order**)	NUMBER []	Returns a NUMBER [] with elements in **l** sorted in specified order. Available orders are **ASC** (for ascending order) and **DESC** (for descending order). Example: **`sort([2, 4, 3, 1], ASC)`** returns **[1, 2, 3, 4]**.
textOnNumberList(number list **numbers**, string **text_expression**)	NUMBER [] STRING	Returns a STRING [] resulting of evaluating **text_expression** against each of the numeric values in argument **numbers**. Argument **text_expression** is an expression that returns a **string**, where **^** represents each numeric value in argument **numbers**. Example: **`textOnNumberList([1, 2, 3, 4, 5], substring("smile", 0, ^))`** returns string list **["s", "sm", "smi", "smil", "smile"]**.

mathOnNumberList(number list **numbers**, number **math_time_expression**)	NUMBER []	Returns a NUMBER [] resulting of evaluating **math_time_expression** against each of the numeric values in argument **numbers**. Argument **math_time_expression** is a math/time expression, where **^** represents each numeric value in argument **numbers**. Example: **`mathOnNumberList([1, 2, 3, 4, 5], ^ * 2)`** returns number list **[2, 4, 6, 8, 10]**.

String lists

Overview

The **String list** **data type** is an ordered list of strings. This data type is returned, among others, by functions that return values of string fields in a selection of issues (**linked issues**, **sub-tasks**, and **subsets**).

Fixed values

A **string list** can also be written in literal form using the following format: **[*string*, *string*, ...]**.

> **Example**
>
> A number list with 5 elements: **`["Blue", "Green", "Yellow", "Orange", "Red"]`**

String list functions

The following functions are intended to build expressions that return **string lists, strings** or **numbers**.

Function	Input	Returned value
filterByCardinality (string list l**, comparison operator** operator**, number** n**)**	STRING [] NUMBER	Returns a STRING [] in **l** whose cardinality (i.e., the number of times it appears in list **l**) satisfies the comparison **cardinality operator n**. Available comparison operators: **=, !=, <, <=, >** and **>=** . Example: **`filterByCardinality(["tiger", "tiger", "tiger", "tiger", "lion", "lion", "lion", "cat", "cat", "lynx"], <, 3)`** returns **`["cat", "lynx"]`** . Example: **`filterByCardinality(fieldValue(%{...components}, subtasks()), =, count(subtasks()))`** returns a list with the **Components** present in all sub-tasks, i.e., those components common to all sub-tasks of current issue.
filterByValue (string list l**, comparison operator** operator**, string** s**)**	STRING [] STRING	Returns a STRING [] in **l** satisfying the comparison **string_in_list operator s**. Example: **`filterByValue(["John", "Robert", "Kevin", "Mark"], ~, "r")`** returns the list of string containing substring **"r"**. i.e., **`["Robert", "Mark"]`**

filterByPredicate(string list **l**, boolean expression **predicate**)	STRING [] BOOLEAN	Returns a STRING [] in **l** that validate **predicate**. Argument **predicate** is a boolean expression, where **^%** is used for referencing string values in argument **l**. Example: **`filterByPredicate(["book", "rose", "sword"], length(^%) > 4)`** returns **`["sword"]`** . Example: **`filterByPredicate(["book", "rose", "sword"], ^% in %{...summary} OR ^% in %{...description})`** returns a list with those strings in first argument that also appear in issue **Summary** or **Description**.
append(string list **l**, string list **m**)	STRING []	Returns a STRING [] with all strings in arguments **l** and **m**. Duplicated string may appear in output. Use function **union**(l, m) instead, if you want to avoid repetitions. Example: **`append(["blue", "red", "green"], ["red", "green", "yellow"])`** returns **`["blue", "red", "green", "red", "green", "yellow"]`** . Example: **`append(fieldValue(%{...fixVersions}, subtasks()), fieldValue(%{...fixVersions}, linkedIssues("is blocked by")))`** returns a string list with **Fix Version/s** of sub-tasks and blocking issues.
union(string list **l**, string list **m**)	STRING []	Returns a STRING [] with all strings in argument **l** or in argument **m** without duplicated strings. Example: **`union(["blue", "red", "green"], ["red", "green", "yellow"])`** returns **`["blue", "red", "green", "yellow"]`** . Example: **`union(fieldValue(%{...fixVersions}, subtasks()), fieldValue(%{...fixVersions}, linkedIssues()))`** returns the list of **Fix Version/s** selected among all sub-tasks and linked issues.
except(string list **l**, string list **m**)	STRING []	Returns a STRING [] with all strings in argument **l** which are not in argument **m**. Duplicated strings in **l** may appear in output. Use function **distinct()** to remove them if you need to. Example: **`except(["blue", "red", "green", "black"], ["red", "green", "yellow"])`** returns **`["blue", "black"]`** . Example: **`except(fieldValue(%{...fixVersions}, subtasks()), fieldValue(%{...fixVersions}, linkedIssues()))`** returns the list of **Fix Version/s** in sub-tasks and not in linked issues.
intersect(string list **l**, string list **m**)	STRING []	Returns a STRING [] with all strings in argument **l** and **m** simultaneously. Example: **`intersect(["blue", "red", "green", "black"], ["red", "green", "yellow"])`** returns **`["red", "green"]`** . Example: **`union(fieldValue(%{...fixVersions}, subtasks()), fieldValue(%{...fixVersions}, linkedIssues()))`** returns the list of **Fix Version/s** common to sub-tasks and linked issues.
distinct(string list **l**)	STRING []	Returns a STRING [] with all strings in list **l** without any duplication. Example: **`distinct(["blue", "green", "yellow", "blue", "yellow"])`** returns **`["blue", "green", "yellow"]`** . Example: **`distinct(fieldValue(%{...assignee}, subtasks()))`** returns the list of assignees to sub-tasks, with only one occurrence per user, although a user may have more than one sub-task assigned.

count(string list **l**)	STRING []	Returns the NUMBER of string values in **l**. Example: **`count(["blue", "red", "blue", "black"])`** returns **4** . Example: **`count(distinct(fieldValue(%{... components}, subtasks())))`** returns the number of **Components** selected among all sub-tasks.
count(string **s**, string list **l**)	STRING STRING []	Returns the NUMBER of times **s** appears in **l**. Example: **`count("blue", ["blue", "blue", "red", "red", "blue", "green"])`** returns **3** .
first(string list **l**)	STRING []	Returns the first element in STRING list **l**, or `null` if **l** is an empty list. Example: **`first(["blue", "red", "green"])`** returns **`"blue"`** .
last(string list **l**)	STRING []	Returns the first element in STRING list **l**, or `null` if **l** is an empty list. Example: **`last(["blue", "red", "green"])`** returns **`"green"`** .
nthElement(string list **l**, number **n**)	STRING [] NUMBER	Returns element at position **n** in STRING list **l**, where **n >= 1** and **n <= count(l)**. Returns `null` if **n** is greater than the number of elements in **l**. Example: **`nthElement(["blue", "red", "green"], 2)`** returns **`"red"`** .
getMatchingValue(string **key**, string list **key_list**, string list **value_list**)	STRING STRING []	Returns STRING value in **value_list** that is in the same position as string **key** is in **key_list**, or in case **key** doesn't exist in **key_list** and **value_list** has more elements than **key_list**, the element of **value_list** in position `count`**`(key_list) + 1`** . Example: **`getMatchingValue("Spain", ["USA", "UK", "France", "Spain", "Germany"], ["Washington", "London", "Paris", "Madrid", "Berlin"])`** returns **`"Madrid"`** .
getMatchingValue(string **key**, string list **key_list**, string list **value_list**)	STRING STRING []	Returns STRING value in **value_list** that is in the same position as numeric **key** is in **key_list**, or in case **key** doesn't exist in **key_list** and **value_list** has more elements than **key_list**, the element of **value_list** in position `count(key_list) + 1`. Example: **`getMatchingValue(8, [2, 4, 6, 8, 10], ["Washington", "London", "Paris", "Madrid", "Berlin"])`** returns **`"Madrid"`** .
sublist(string list **l**, number **indexFrom**, number **indexTo**)	STRING [] NUMBER	Returns a STRING [] with elements in **l** from **indexFrom** index to **indexTo** index. Having **indexFrom >= 1** and **indexFrom <= count(l)** and **indexTo >= 1** and **indexTo <= count(l)** and **indexFrom <= indexTo**. Example: **`sublist(["red", "green", "blue", "purple", "white"], 2, 4)`** returns **`["green", "blue", "purple"]`** .
indexOf(string **element**, string list **l**)	STRING STRING []	Returns the index NUMBER of string **element** in string list **l**. **Zero** is returned when **element** is not found in **l**. Example: **`indexOf("blue", ["red", "blue", "green"])`** returns **2** .
sort(string list **l**, order)	STRING []	Returns a STRING [] with elements in **l** lexicographically ordered. Available orders are **ASC** (for ascending order) and **DESC** (for descending order). Example: **`sort(["red", "blue", "green"], ASC)`** returns **`["blue", "green", "red"]`** .

textOnStringList(string list **strings**, string **text_expression**)	STRING []	Returns a STRING [] resulting of evaluating **text_expression** against each of the strings in argument **strings**. Argument **text_expression** is an expression that returns a string, where **^%** represents each string in argument **strings**. Example: `textOnStringList(["albert", "riCHard", "MARY"], capitalizeWordsFully(^%))` returns `["Albert", "Richard", "Mary"]` .
mathOnStringList(string list **strings**, number **math_time_expression**)	STRING []	Returns a NUMBER [] resulting of evaluating **math_time_expression** against each of the issues in argument **issues**. Argument **math_time_expression** is a math/time expression, where **^%** represents each string in argument **strings**. Example: `mathOnStringList(["a", "ab", "abc", "abcd", "abcde"], length(^%))` returns `[1, 2, 3, 4, 5]` .

Examples

Input	Output
`["red", "blue", "green"]`	A string list with the names of 3 colors
`fieldValue(%{...summary}, subtasks())`	Returns the list of s**ummaries of sub-tasks** of the current issue
`toStringList(%{...components})`	Returns a list with the **names of the components** of the current issue.
`distinct(toStringList(toString(fieldValue(%{...components}, subtasks())), ","))`	Returns a string list with all the components present in the sub-tasks of the current issue **without duplicates**.

List operators

General Information

There are **three** different data types that return **lists**. i.e., types that are based on lists, or ordered collections of elements.

These data types are:

- **Issue lists** ISSUE []
- **Number lists** NUMBER []
- **String lists** STRING []

List Operators

There are **four available operators** for working on **list-based** data types:

Operator	Behavior	Examples

l **APPEND** m	Returns a list with elements in **l** followed by elements in **m**, therefore the number of elements is the sum of the number of elements in **l** and **m**.
	Order is respected. It may contain repeated elements.
		`[1, 2, 3] APPEND [3, 4, 4] = [1, 2, 3, 3, 4, 4]`
		
		`["blue", "red", "red"] APPEND ["red", "green"] = ["blue", "red", "red", "red", "green"]`
		
		`subtasks() UNION subtasks()` returns a list containing twice all the sub-tasks of current issue.

l **`UNION`** m	Returns a list with elements in **l** and elements **m** without repetitions. Order is respected.	**`[1, 2, 3] UNION [3, 4, 4] = [1, 2, 3, 4]`** **`["blue", "red", "red"] UNION ["red", "green"] = ["blue", "red", "green"]`** **`linkedIssues() UNION subtasks()`** returns a list with linked issues and sub-tasks of current issue without repetitions.
l **`INTERSECT`** m	Returns a list with the elements present in both lists simultaneously. Returned list doesn't contain element repetitions. Order is respected.	**`[1, 1, 2, 3] INTERSECT [1, 3, 5] = [1, 3]`** **`["red", "blue", "blue"] INTERSECT ["blue", "yellow", "yellow"] = ["blue"]`** **`linkedIssues() INTERSECT subtasks()`** returns a list with those sub-tasks which are also linked to current issue.
l **`EXCEPT`** m	Returns a list with elements in **l** which are not present in list **m**. Returned list doesn't contain element repetitions. Order is respected.	**`[1, 2, 2, 3, 3] EXCEPT [2, 5, 6] = [1, 3]`** **`["red", "red", "blue", "blue", "green"] EXCEPT ["blue", "yellow"] = ["red", "green"]`** **`linkedIssues() EXCEPT subtasks()`** returns a list with linked issues which are not sub-tasks of current issue.

- **l** and **m** are both lists of the same data type: **number**, **string** or **issues**.
- All operators are case insensitive, i.e., they can also be written in lower case: **`append`**, **`union`**, **`intersect`** and **`except`** .
- There are 4 equivalent and homonym functions available for each type of list, and its behavior is exactly equivalent to that of its corresponding operator. This way, you can choose to use operators or functions according to your preference. Although operators yield shorter expressions and with fewer parentheses, the usage of functions produces a more functional consistent syntax

Precedence Order and Associativity

OPERATORS	PRECEDENCE	ASSOCIATIVITY
l **`INTERSECT`** m	1 (highest)	Left-to-Right
l **`UNION`** m, l **`EXCEPT`** m, l **`APPEND`** m	2 (lowest)	Left-to-Right

Selectable fields

Overview

Selectable fields are fields with a **limited domain or set of options** or **possible values**.

These fields includes:

- **Select**

- **Multi Select**
- **Radio Button**
- **Security Level**
- **Checkboxes**
- **Components**
- **Versions**
- **Multi User Picker**
- **Multi Group Picker**
- **Issue Pickers**
- **Attachments**
- **Labels**

Available functions

Function	Input	Returned value
numberOfSelectedItems(%{...somefield}) : number	FIELD	Returns the NUMBER of selected items in select or multiselect field with field code **%{...somefield}**.
numberOfAvailableItems(%{...somefield}) : number	FIELD	Returns the NUMBER of available options in select or multiselect field with field code **%{...somefield}**. It's equivalent to **`count(availableItems(%{...somefield}))`** . Disabled options are discarded.
availableItems(%{...somefield}) : string list	FIELD	Returns a STRING [] with available options in select or multiselect field with field code **%{...somefield}**. Disabled options are discarded. Example: **`availableItems(%{00103})`** returns a string list with all **security levels** available for the project and current user.
availableItems(**%{...somefield}**, string **option**) : string list 0	FIELD	Returns a STRING [] with all available child options in cascading or multilevel cascading field with ID **%{...somefield}**, and for option parent **option**. In the case of multilevel cascading fields, a comma separated list of options should be entered. Disabled options are discarded.
allAvailableItems(%{...somefield}) : string list	FIELD	Returns a STRING [] with all available options in select or multiselect field with field code **%{...somefield}**. Disabled options are included. Example: **`availableItems(%{00103})`** returns a string list with all **security levels** available for the project and current user.
allAvailableItems(**%{...somefield}**, string **option**) : string list	FIELD	Returns a STRING [] with all available child options in cascading or multilevel cascading field with ID **%{...somefield}**, and for option parent **option**. In the case of multilevel cascading fields, a comma separated list of options should be entered. Disabled options are included.

Users, groups and roles

Overview

The expression parser offers multiple functions to manage user-, group- and role-related information.

Available functions

Function	Input	Returned value

isInGroup(string **user_name**, string **group_name**)	STRING	Checks if a user is in a group. BOOLEAN Argument **user_name** can also be a **comma separated list** of **user names**, **group names** or **role names**. In that case the function will return **true** only if **all users** in the list, groups of the list, and in the roles of the list, are in the group in the second argume{nt. **Example** **`isInGroup(%{...assignee}, "jira-developers")`** returns true if **Assignee** in in group **jira-developers.**
isInRole(string **user_name**, string **role_name**)	STRING	Checks if a user or group of users plays a role in current project. BOOLEAN Argument **user_name** can also be a **comma separated list** of **user names**, **group names** or **role names**. In that case the function will return **true** only if **all users** in the list, groups of the list, and in the roles of the list, are in project role in the second argument, for current project. **Example** **`isInRole(%{...reporter}, "Testers")`** returns true in **Reporter** is in project role **Testers**.
isInRole(string **user_name**, string **role_name**, string **project_key**)	STRING	Checks if a user or group of users plays a role in a certain project. BOOLEAN Argument **user_name** can also be a **comma separated list** of **user names**, **group names** or **role names**. In that case the function will return **true** only if **all users** in the list, groups of the list, and in the roles of the list, are in role in the second argument, for the project in the third argument. **Example** **`isInRole(%{...currentUser}, "Developers", "CRM")`** returns true in **Current user** is in project role **Developers** in project with key **"CRM".**
isActive(string **user_name**)	STRING	Checks if a user is active. BOOLEAN Argument **user_name** can also be a **comma separated list** of **user names**, **group names** or **role names**. In that case the function will return **true** only if **all users** in the list, groups of the list, and in the roles of the list, are active. **Example** **`isActive(%{...componentLeads})`** returns true if all users who are **component leaders** in current project are active.

userFullName(string **user_name**)	STRING	Returns a STRING with the full name of the user in argument **user_name**. Argument **user_name** is a string with a user name, not to be confused with user full name. **Example** **`userFullName(%{...currentUser})`** returns the user's full name of current user.
userFullName(string list **user_names**)	STRING []	Returns a STRING [] with the full names of the users in argument **user_names**. Argument **user_names** is a string list with user names, not to be confused with users full names. **Example** **`userFullName(toStringList(%{...watchers}))`** returns a list with the users full names of current issue's watchers.
userEmail(string **user_name**)	STRING	Returns a STRING with the email of the user in argument **user_name**. Argument **user_name** is a string with a user name, not to be confused with user full name. **Example** **`userEmail(%{...currentUser})`** returns the email of current user.
userEmail(string list **user_names**)	STRING []	Returns a STRING [] with the emails of the users in argument **user_names**. Argument **user_names** is a string list with a user names, not to be confused with users full names. **Example** **`userEmail(toStringList(%{...watchers}))`** returns a list with the emails of current issue's watchers.
fullNameToUser(string **fullName**)	STRING	Returns a STRING with the name of a user whose full name is equal to argument **fullName**. Returned value is a string with a **user name**.
usersWithEmail(string **email**)	STRING	Returns a STRING [] with the **user names** of those users with emails equal to argument **email**. In case that only one user is expected, function **first**(string list) can be used to extract a string with its user name.
userProperty(string **propertyName**, string **userName**)	STRING	Returns the STRING value of the user property with name **propertyName** which belongs to user with user name **userName**. If the user doesn't have the property, `""` will be returned.

userProperty(string **propertyName**, string list **userNames**)	STRING	Returns the STRING [] of values of the user property with name **propertyName** in all the users whose names are contained in **userNames**. The output will contain as many strings as users have the property set.
usersInRole(string **projectRoleName**)	STRING	Returns the STRING [] of **user names** (not be confused with **full user name**) of those active users playing project role with name **projectRoleName** in current issue's project. Parameter **projectRoleName** can be a comma separated list of project role names, returning the users that play any of the project roles.
usersInRole(string **projectRoleName**, string **projectKey**)	STRING	Equivalent to the previous function that returns a STRING [] with extra argument **projectKey** for selecting the project argument **projectRoleName** refers to.
usersInGroup(string **groupName**)	STRING	Returns the STRING [] of **user names** of those active users in group with name **groupName**. Parameter **groupName** can be a comma separated list of group names, returning the users that belong to any of the groups.
rolesUserPlays(string **userName**)	STRING	Returns the STRING [] of **role names** of those project roles the user with name **userName** plays in current project. Parameter **userName** can also be a comma separated list of **user names, group names** and **project role names**, returning the list of project roles for those users represented by input argument.
rolesUserPlays(string **userName**, string **projectKey**)	STRING	Returns the STRING [] of **role names** of those project roles the user with name **userName** plays in project with key **projectKey**. Parameter **userName** can also be a comma separated list of **user names, group names** and **project role names**, returning the list of project roles for those users represented by input argument.
groupsUserBelongsTo(string **userName**)	STRING	Returns the STRING [] of **group names** of those groups the user with name **userName** belongs to. Parameter **userName** can also be a comma separated list of **user names, group names** and **project role names**, returning the list of project roles for those users represented by input argument.
defaultUserForRole(string **projectRoleName**)	STRING	Returns the STRING of the **user name** of the **Assign to project role** project role with name **projectRoleName** in current issue's project, or `""` if no default user is defined for the project role.
defaultUserForRole(string **projectRoleName**, string **projectKey**)	STRING	Equivalent to the previous STRING function but with extra argument **projectKey** for selecting the project argument **projectRoleName** refers to.
lastAssigneeInRole(string **projectRoleName**)	STRING	Returns the STRING of **user name** of the last user who had current issue assigned, and currently plays project role with name **projectRoleName** in current issue's project, or `""` if current issue was never assigned to a user currently in the project role.
lastAssigneeInRole(string **projectRoleName**, string **issueKey**)	STRING	Returns the STRING of **user name** of the last user who had issue with key **issueKey** assigned, and currently plays project role with name **projectRoleName** in current issue's project, or `null` if current issue was never assigned to a user currently in the project role.

leastBusyUserInRole(string **projectRoleName**)	STRING	Returns the name of the active user playing project role with name **projectRoleName** in current issue's project, and has the lower number of issues with resolution empty assigned; or `""` if there isn't any user in the project role. Parameter **projectRoleName** can be a comma separated list of project role names, returning the least busy users among the project roles. **Example** `leastBusyUserInRole("Developers")` returns the STRING user playing role **Developers** in current project with the least number of unresolved issues in all the Jira instance assigned.
leastBusyUserInRole(string **projectRoleName**, string **projectKey**)	STRING	Equivalent to the previous function but with extra argument **projectKey** for selecting the project argument **projectRoleName** refers to. Example: `leastBusyUserInRole("Developers", "CRM")` returns STRING of the user playing role **Developers** in project with key **CRM** with the least number of unresolved issues in all the Jira instance assigned.
leastBusyUserInRole(string **projectRoleName**, string **projectKey**, string **jqlQuery**)	STRING	Equivalent to the previous function but with extra argument **jqlQuery**, used for restricting the issues to be considered to pick the least busy user as a STRING. **Example** `leastBusyUserInRole("Developers", %{...projectKey}, "project = " + %{...projectKey})` returns the user playing role **Developers** in current project, with the least number of unresolved issues in current project assigned.
nextUserInGroup(string **groupName**, string **queueName**)	STRING	Returns the STRING name of the next active user in group with name **groupName**, for a round-robin queue with name **queueName**. The string **queueName** is an arbitrary name. The queue is automatically created the first time a queue is used in a function call. Each time the function is called on the same pair of arguments `(group, queue)` , a different user in the group is returned. The queue can be used in different transitions of the same or different workflows within the same Jira instance. `null` is returned if **group** is empty. **Example** `nextUserInGroup("jira-developers", "code-review-queue")` returns the username of the next user in group **jira-developers** for round-robin queue **code-review-queue**. Each time the function is called with the same pair of arguments, a different username is returned.

Versions

Overview

The expression parser offers multiple functions to retrieve **version related** field values.

Available functions

Function	Input	Returned value
unreleasedVersions()		Returns a STRING [] with unreleased version names of current issue's project. Returned versions may be archived. Example: **`toStringList(%{...versions}) any in unreleasedVersions()`** validates that at least one affected version is unreleased.
unreleasedVersions(string **projects**)	STRING	Returns a STRING [] with unreleased version names of projects in argument **projects**. Returned versions may be archived. Arguments **projects** is a comma separated list of **project keys** or **project names**.
releasedVersions()		Returns a STRING [] with released version names of current issue's project. Returned versions may be archived. Example: **`toStringList(%{...fixVersions}) in releasedVersions()`** validates that all fixed versions are released.
releasedVersions(string **projects**)	STRING	Returns a STRING [] with released version names of projects in argument **projects**. Returned versions may be archived. Arguments **projects** is a comma separated list of **project keys** or **project names**. Example: **`toStringList(^%{...fixVersions}) in releasedVersions(^%{...projectKey})`**validates that all fixed versions of a foreign issue are released.
releaseDates(string **versions**)	STRING	Returns a NUMBER [] with the release dates for versions in string **versions** for current issues project. Parameter **versions** is a comma separated list of version names. Example: **`releaseDates(%{...fixVersions})`** returns the list of release dates for **Fix Version/s**.
releaseDates(string **versions** , string **projects**)	STRING	Returns a NUMBER [] with the release dates for versions in string **versions** for projects in parameter **projects**. Parameter **versions** is a comma separated list of version names. Parameter **projects** is a comma separated list of project keys or project names. Example: **`releaseDates(%{...versions}, "CRM")`** returns the list of release dates for affected versions for project with key **"CRM"**. .
startDates(string **versions**)	STRING	Returns a NUMBER [] with the start dates for versions in string **versions** for current issues project. Parameter **versions** is a comma separated list of version names. Example: **`startDates(%{...fixVersions})`** returns the list of start dates for fixed versions.
startDates(string **versions**, string **projects**)	STRING	Returns a NUMBER [] with the start dates for versions in string **versions** for projects in parameter **projects**. Parameter **versions** is a comma separated list of version names. Parameter **projects** is a comma separated list of project keys or project names. Example: **`startDates(%{...versions}, "CRM")`** returns the list of start dates for affected versions for project with key *"* **CRM** *"*.
archivedVersions()		Returns a STRING [] with released version names of current issue's project. Returned versions may be archived.
archivedVersions(string **projects**)	STRING	Returns a STRING [] with released version names of projects in argument **projects**. Returned versions may either released or unreleased. Arguments **projects** is a comma separated list of **project keys** or **project names**.

latestReleasedVersion()		Returns STRING with the name of the latest released version in current issue's project. Example: **`latestReleasedVersion() in archivedVersions()`** validates that the latest released version in current issue's project is archived.
latestReleasedVersion(string **projects**)	STRING	Returns STRING [] with the name of the latest released version among projects in argument **projects**. Returned versions may either released or unreleased. Arguments **projects** is a comma separated list of **project keys** or **project names**.
latestReleasedUnarchivedVersion(string **projects**)	STRING	Returns STRING [] with the name of the latest released version excluding archived ones for projects in argument **projects**. Returned versions may either released or unreleased. Arguments **projects** is a comma separated list of **project keys** or **project names**.
earliestUnreleasedVersion()		Returns STRING with the name of the earliest unreleased version in current issue's project. Example: **`earliestUnreleasedVersion() not in archivedVersions()`** validates that earliest unreleased version in current issue's project is not archived.
earliestUnreleasedVersion(string **projects**)	STRING	Returns STRING [] with the name of the earliest unreleased version among projects in argument **projects**. Returned versions may either released or unreleased. Arguments **projects** is a comma separated list of **project keys** or **project names**.
earliestUnreleasedUnarchivedVersion()		Returns STRING with the name of the earliest unreleased version in current issue's project excluding archived ones.
earliestUnreleasedUnarchivedVersion(string **projects**)	STRING	Returns STRING [] with the name of the earliest unreleased version excluding archived ones for projects in argument **projects**. Returned versions may either released or unreleased. Arguments **projects** is a comma separated list of **project keys** or **project names**.
unreleasedVersionsBySequence() Available since version 1.1.0		Returns a STRING [] with the unreleased versions **in the current project** with the default order. Only non-archived versions are returned. The first version in the list is the lowermost version in the version table.
releasedVersionsBySequence() Available since version 1.1.0		Returns a STRING [] with the released versions **in the current project** with the default order. Only non-archived versions are returned. The first version in the list is the lowermost version in the version table.

Historical field values

Overview

The expression parser offers multiple functions to retrieve historical field values.

Functions for accessing historical field values are available for the following fields:

- All **Custom Fields**
- Summary
- Description
- Assignee
- Reporter
- Due date

- Issue status
- Priority
- Resolution
- Environment
- Fix version/s
- Affects version/s
- Labels
- Components
- Security level

Available functions

Function	Input	Returned value
previousValue(%{...somefield})	FIELD	Returns a STRING with the previous value of a field for current issue. It will return **`null`** if field was previously uninitialized.
previousValue({...somefield})	FIELD	Returns a NUMBER with the previous value of a **numeric** or **date field** for current issue. It will return **`null`** if field was previously uninitialized.
previousValue(%{...somefield.i})	FIELD	Returns a STRING with the previous value of a **cascading** or **multi-cascading select field** for current issue **at level i** (with root level = 0). It will return **`null`** if field was previously uninitialized.
fieldHistory(%{...somefield})	FIELD	Returns a STRING [] with all the values that a field has ever had in the past for current issue. Values appear in the list in ascending ordered by setting time, i.e., older value has index `1`, and most recent value has index **`count(string_list)`** . Uninitialized field statuses are represented by **empty strings**.
fieldHistory({...somefield})	FIELD	Returns a NUMBER [] with all the values that a **numeric** or **date-time field** has ever had in the past for current issue. Values appear in the list in ascending ordered by setting time, i.e., older value has index `1`, and most recent value has index **`count(number_list)`** . Uninitialized field statuses are not represented.
fieldHistory(%{...somefield.i})	FIELD	Returns a STRING [] with all the values that a **cascading** or **multi-cascading select field** has ever had in the past **for level i** (with root level = 0) in current issue. Values appear in the list in ascending ordered by setting time, i.e., older value has index `1`, and most recent value has index **`count(string_list)`** . Uninitialized field statuses are represented by empty strings.
hasChanged(%{...somefield})	FIELD	Returns BOOLEAN **true** only if field has changed in current transition. Function **hasChanged(field_code)** is used when we set a validation that is incompatible with a condition in a same transition, typically when validating a value entered in the transition screen. When Jira evaluates the validations in a transition, it also reevaluates the conditions, and if they are not satisfied an **`Action X is invalid`** error message is shown and the transition is not executed. **Example:** Let's suppose we have a boolean condition like **`{...duedate} = null`** (i.e., **Due date = null**) in a transition, so that it's only shown when **Due date** is empty. This transition also has a transition screen containing field **Due date**, and a boolean validation **`{...duedate} != null`** , in order to make **Due date** required in the transition. The configuration described above will not work, since both condition and validation are mutually incompatible. We can fix it replacing the boolean condition with **`{...duedate} = null OR hasChanged(%{...duedate})`** .
hasChanged({...somefield})	FIELD	Returns BOOLEAN **true** only if **numeric** or **date-time field** field has changed in current transition.

hasChanged({...somefield.i})	FIELD	Returns BOOLEAN **true** only if **cascading select field** has changed **for level i** (with root level = 0) in current transition.

Miscellaneous

Overview

The expression parser offers multiple functions that **cannot easily be categorized**.

A comprehensive list can be found below.

Available functions

Function	Input	Returned value
projectProperty(string **property_name**)	STRING	Returns a STRING with the value of project property with name **property_name** in current issue's project. **`null`** is returned if project property doesn't exist. Example: **`projectProperty("maxNumberOfReopenings")`** returns **`"3"`** , provided there is a string **`{maxNumberOfReopenings=3}`** in the description of current issue's project.
projectProperty(string **property_name**, string **project_key**)	STRING	Returns a STRING with the value of project property with name **property_name** in project with key **project_key**. **`null`** is returned if project property doesn't exist. Example: **`projectProperty("maxNumberOfReopenings", "CRM")`** returns **`"3"`** , provided there is a string **`{maxNumberOfReopenings=3}`** in the description of project with key **CRM**.
projectPropertyExists(string **property_name**)	STRING	Returns BOOLEAN **true** only if there is a project property with name **property_name** in current issue's project, i.e., if project's description contains a string like **{property_name=value}**. Example: **`projectPropertyExists("maxNumberOfReopenings")`** returns **`true`** only if there is a string like **`{maxNumberOfReopenings=x}`** in the description of current issue's project.
projectPropertyExists(string **property_name**, string **project_key**)	STRING	Returns BOOLEAN **true** only if there is a project property with name **property_name** in project with key **project_key**. **Example** **`projectPropertyExists("maxNumberOfReopenings", "CRM")`** returns **`true`** only if there is a string like **`{maxNumberOfReopenings=x}`** in the description of project with key `CRM`.
isAClone()		Returns BOOLEAN **true** only if current issue is a clone of another issue. An issue is a clone of another issue if it's being created by **Jira "Clone" operation**, or has **issue links of type "clones"**. This function if useful for bypassing validations in transition **Create Issue** when the issue is being created by a clone operation.

allComments()		Returns a STRING [] with all the comments in current issue in ascension order by creation date.
allComments(string **issue_keys**)	STRING	Returns a STRING [] with all the comments in issues with keys in **issue_keys**, in order of appearance in **issue_keys**, and by creation date in ascension order. Argument **issue_keys** is a comma separated list of issue keys. **Example** **`allComments(%{...parentIssuekey})`** returns parent issue's comments.
allComments(issue list **l**)	ISSUE []	Returns a STRING [] with all the comments in issues in **l**, in order of appearance in **l**, and by creation date in ascension order. Example: **`allComments(subtasks())`** returns all the comments in all the sub-tasks of current issue.
allCommenters()		Returns a STRING [] with the user names of comment authors and updaters in current issue, in ascension order by commenter's actuation time. The same user appears in the output as many times as the comments the user created and updated.
allCommentCreators()		Returns a STRING [] with the user names of comment creators in current issue, in ascension order by commenter's actuation time. A same user appears in the output as many times as comments has created. For anonimous comments an empty string (`""`) is returned.
allCommentCreators(string **issue_keys**)	STRING	Returns a STRING [] with the user names of comment creators in issues with keys in **issue_keys**, in order of appearance in **issue_keys**, and in ascension order by commenter's actuation time. A same user appears in the output as many times as comments has created. For anonymous comments an empty string (`""`) is returned.
allCommentCreators(string list **l**)	STRING []	Returns a STRING [] with the user names of comment creators of issues in **l**, in order of appearance in **l**, and in ascension order by commenter's actuation time. A same user appears in the output as many times as comments has created. For anonimous comments an empty string (`""`) is returned.
allCommenters(string **issue_keys**)	STRING	Returns a STRING [] with the user names of comment authors and updaters of issues with keys in **issue_keys**, in order of appearance in **issue_keys**, and in ascension order by commenter's actuation time. Argument **issue_keys** is a comma separated list of issue keys. **Example** **`allComments(%{...parentIssuekey})`** returns a string list with the user names of comment authors of parent issue.
allCommenters(issue list **l**)		Returns a STRING [] with the user names of comment authors and updaters of issues in **l** in ascension order by actuation time, in order of appearance in **l**, and in ascension order by commenter's actuation time. Example: **`allCommenters(linkedIssues("is blocked by"))`** returns a list with all the commenters and comment updaters for linked issues blocking current issue.

usersWhoTransitioned(string **origin_status**, string **destination_status**)	STRING	Returns a STRING [] with the names of the users who transitioned current issue from **origin_status** to **destination_status**, order ascending by time. An empty string as argument is interpreted as **any status**. **Example** **`last(usersWhoTransitioned("Open", "In Progress"))`** returns the name of the user who executed transition **"Start Progress"** more recently.
usersWhoTransitioned(string **origin_status**, string **destination_status**, string **issue_key**)	STRING	Returns a STRING [] with the names of the users who transitioned current issue from **origin_status** to **destination_status**, order ascending by time. An empty string as argument is interpreted as **any status**. **Example** **`count(usersWhoTransitioned("Open", "In Progress", %{...parentIssuekey}))`** returns the number of times transition **"Start Progress"** has been executed in parent issue.
timesOfTransition(string **origin_status**, string **destination_status**)	STRING	Returns a NUMBER [] with the times when current issue was transitioned from **origin_status** to **destination_status**, order ascending by time. An empty string as argument is interpreted as **any status**. **Example** **`last(timesOfTransition("", "Resolved"))`** returns the most recent time when the issue was resolved.
timesOfTransition(string **origin_status**, string **destination_status**, string **issue_key**)	STRING	Returns a NUMBER [] with the times when issue with key **issue_key** was transitioned from **origin_status** to **destination_status**, order ascending by time. An empty string as argument is interpreted as **any status**. **Example** **`first(usersWhoTransitioned("Closed", "", %{...parentIssuekey}))`** returns the first time when parent issue was reopened.
componentLeader(string **component_name**)	STRING	Returns the **user name** of the component lead with name **component_name** in current issue's project as STRING. This function also admits a comma separated list of components, and returns a comma separated list of **user names**. Output will contain repeated user names if a same user is leader of more than one component. **Example** **`componentLeader(%{...components})`** returns a comma separated list with the user names of the leaders of current issue's components.

componentLeader(string **component_name**, string **project_key**)	STRING	Returns a the **user name** of the component lead with name **component_name** in project with key **project_key as** STRING. This function also admits a comma separated list of components, and returns a comma separated list of **user names**. Output will contain repeated user names if a same user is leader of more than one component. **Example** **`componentLeader("Web Portal", "CRM")`** returns the user name of the leader of the component with name **Web Portal** in project with key **CRM**.
issueIDFromKey(string **issue_key**)	STRING	Returns a STRING of the **internal ID** of issue with key **issue_key**. This function also admits a comma separated list of issue keys, and returns a comma separated list of **internal IDs**. **Example** **`issueIDFromKey("CRM-1")`** returns **`"10001"`** .
issueKeyFromID(string **issue_ID**)	STRING	Returns a STRING of the **issue key** of issue with **internal ID issue_ID**. This function also admits a comma separated list of **issue IDs**, and returns a comma separated list of **issue keys**. **Example** **`issueIDFromKey("10001")`** returns **`"CRM-1"`** .
projectKeys()		Returns a STRING [] with all the *project keys* in the JIRA instance.
projectKeys(string **category)**	STRING	Returns a STRING [] with the *project keys* of those projects that belong to project category with name **category.**
projectName(string **project_key)**	STRING	Returns a STRING with the **name** of the project with key **project_key**.
projectCategory(string **project_key)**	STRING	Returns a STRING with the **category** of the project with key **project_key**.

Functions to temporarily store and retrieve values

Function	Returned value
setBoolean(string **variable_name**, boolean **value**) : boolean	Creates a variable named **variable_name** for storing a boolean value, and assigns it a **value**, which is also returned in order to be used within an expression. Example: **`setBoolean("myBoolean",true)`**
getBoolean(string **variable_name**) : boolean	Returns the value stored in a boolean variable named **variable_name**, which was previously created using the **setBoolean()** function. Example: **`getBoolean("myBoolean")`**

setNumber(string **variable_name**, number **value**) : number	Creates a variable named **variable_name** for storing a number, and assigns it a **value**, which is also returned in order to be used within an expression. Example: **`setNumber("myNumber",100)`**
getNumber(string **variable_name**) : number	Returns the value stored in a numeric variable named **variable_name**, which was previously created using the **setNumber()** function. Example: **`getNumber("myNumber")`**
setString(string **variable_name**, string **value**) : string	Creates a variable named **variable_name** for storing a string, and assigns it a **value**, which is also returned in order to be used within an expression. Example: **`setString("myString","Hello World!")`**
getString(string **variable_name**) : string	Returns the value stored in string variable named **variable_name**, which was previously created using the **setString()** function. Example: **`getString("myString")`**
setNumberList(string **variable_name**, number list **value**) : number list	Creates a variable named **variable_name** for storing a number list, and assigns it a **value**, which is also returned in order to be used within an expression. Example: **`setNumberList("myNumberList",[1,2,3])`**
getNumberList(string **variable_name**) : number list	Returns the value stored in number list variable named **variable_name**, which was previously created using the **setNumberList()** function. Example: **`getNumberList("myNumberList")`**
setStringList(string **variable_name**, string list **value**) : string list	Creates a variable named **variable_name** for storing a string list, and assigns it a **value**, which is also returned in order to be used within an expression. Example: **`setStringList("myStringList",["Hello"," World"])`**
getStringList(string **variable_name**) : string list	Returns the value stored in string list variable named **variable_name**, which was previously created using the **setStringList()** function. Example: **`getStringList("myStringList")`**
setIssueList(string **variable_name**, issue list **value**) : issue list	Creates a variable named **variable_name** for storing an issue list, and assigns it a **value**, which is also returned in order to be used within an expression. Example: **`setIssueList("myIssueList",["KEY-1","KEY-2"])`**
getIssueList(string **variable_name**) : issue list	Returns the value stored in issue list variable named variable_name, which was previously created using setIssueList() function. Example: **`getIssueList("myIssueList")`**