Search documentation Search

Appian

Looping Functions

Appian 7.8

The following functions can be used within expressions to perform looping operations that call a rule or function for each item in an array. Toggle Na

- Returns a Boolean Value
  - all()
  - any()
  - none()
- Returns a Single Value or New Array
  - filter()
  - reduce()
  - apply()
  - reject()
  - merge()

The following table summarizes the behavior of each function:

Function	Expression	Result
apply()	apply(fn!isnull, {1, null, 3})	{false, true, false}
reduce()	reduce(fn!sum, 0, {1, 2, 3})	6
reject()	reject(fn!isnull, {1, null, 3})	{1, 3}
filter()	filter(fn!isnull, {1, null, 3})	{null}
any()	any(fn!isnull, {1, null, 3})	true
all()	all(fn!isnull, {1, null, 3})	false
none()	none(fn!isnull, {1, null, 3})	false
merge()	merge({1, 2, 3}, {10, 20, 30})	{[1, 10], [2, 20], [3, 30]}

They simplify array manipulation inside and outside process design and can be used instead of multiple node instances in cases where the result is a sing value or a new array.

See also: Multiple Node Instances and Function Recipes for example functions that demonstrate this.

- The referenced rule or function must have one or more parameters and will be evaluated once for each item in the provided list.
- Rules are referenced using `rule!` with no parentheses and functions are referenced using `fn!`.
  - The functions `if()`, `and()`, `or()`, and `byreference()` cannot be referenced as the function to loop through.
- If the rule/function must iterate over more than one list, use the merge() function to combine the lists.
- An expression may be used as the first parameter in the looping function, but the result of the expression must be a reference to a rule or function. For example, since `if(pv!hasApproved, rule!approvalRuleToRun, fn!rejectFunctionToRun)` returns a rule or function, it can be used as a parameter for a looping function.
- If the rule or function definition changes while a looping function is evaluated, the original definition is preserved throughout the evaluation of all items in the array.

Rules and expression functions with a specified name that are referenced by the looping functions will be found by dependency analysis in the Applicatic Builder and must exist on the target system or in the package for an application import to succeed.

- The rules will be referenced by UUID in the exported XMLs, not by the rule name.
- The expression functions that are referenced by the looping functions are exported by name, not by the UUID (there is no separate UUID for functions).

See also: Dependency Analysis and Application Builder.

For the full list of Appian Functions, see also: Appian Functions

NOTE: The arguments you pass to these functions cannot call the `load()` function.

Best Practices

The following best practices should be observed when using a looping function.

- If you find more than one looping function could be used for your use case, pick the one that would make your expression more readable and easiest to maintain by the next person on your project.
  - For example, `reduce()` is the most versatile looping function and can be used to solve all looping cases. However, it may be easier for your colleague to immediately understand the result of `=filter(rule!myrule, pv!myArray, pv!criteria)` than the result of `=reduce(rule!myrule, {},`

`pv!myArray, pv!criteria)` .

- If the data to loop through comes from a query rule or from any call to an external system, execute the query rule first, save it into an ACP or P (depending on the circumstances), and call the looping function on the variable rather than the query rule itself.
- If the referenced rule is a query rule or from any call to an external system, redefine the query rule to operate on a list if possible.
  - For example, use the "in" operator in your query conditions instead of the "=" operator.
- Instead of a looping function, consider creating a database view for complex data manipulation that can be done more efficiently by the databas
- Many out of the box functions operate on lists already, so you may not need to use a looping function in such cases.
  - For example, isleapyear() function and sum() function.
- If the number of items to loop through could be large, test the performance with a representative data set in development by using the expressi in a script task and reviewing the Duration column in Process Details > Process Nodes tab to see how long it takes to evaluate.
  - A duration approaching 5 seconds should be redesigned.
- For large lists, consider writing a specialized expression plugin that will do the processing rather than using a looping function.

Returns a Boolean Value

The following Looping Functions call a rule or function that returns either true or false for each item in a list and returns a Boolean value based on the values returned for the items in the list.

all()

Calls a rule or function that returns either true or false for each item in list, asks the question, "Do all items in this list yield true for this rule/function?", an returns true if all items in list evaluates to true.

Syntax

all(*predicate*, *list*, [*context*,...])

predicate: (Function, Rule, or Data Type Constructor) Expression that returns a Boolean (true or false).

list: (Any Type Array) List of elements that the predicate iterates through.

context: (Any Type Array) Variable number of parameters passed directly into each predicate evaluation.

Returns

Boolean

Notes

Use `fn!functionName` to reference an expression function and `rule!ruleName` to reference a rule.

Returns false as soon as the returned value of an evaluation yields false; otherwise, returns true.

Empty and null lists yield true.

Serves as an alternative to `and(apply(rule!iseven, {-1,0,1,2}))` .

Examples

Copy and paste an example into the Test Rules interface to see how this works.

Given a function `f(x)` , `all(fn!f, {a, b, c}, v)` returns `and({f(a, v), f(b, v), f(c, v)})` .

`all(rule!iseven,{-1,0,1,2},1)` returns `false`

any()

Calls a rule or function that returns either true or false for each item in list by asking the question, "Do any items in this list yield true for this rule/function?" with the intent to discover if any item(s) yield true.

Syntax

any(*predicate*, *list*, [*context*, ...])

predicate: (Function, Rule, or Data Type Constructor) Expression that returns a Boolean (true or false).

list: (Any Type Array) List of elements that the predicate iterates through.

context: (Any Type Array) Variable number of parameters passed directly into each predicate evaluation.

Returns

Boolean

Notes

Use `fn!functionName` to reference an expression function and `rule!ruleName` to reference a rule.

Returns true as soon as the returned value of an evaluation yields true; otherwise, returns false.

Empty and null lists yield false.

Serves as an alternative to `or(apply(rule!iseven, {-1,0,1,2}))` .

Examples

Copy and paste an example into the Test Rules interface to see how this works.

Given a function `f(x)` , `any(fn!f, {a, b, c}, v))` returns `or({f(a, v), f(b, v), f(c, v)})`

`any(rule!iseven,{-1,0,1,2})` returns `true`

none()

Calls a rule or function that returns either true or false for each item in list by asking the question, "Do all items in this list yield false for this rule/function? with the intent to discover if no items will yield true.

Syntax

none(*predicate*, *list*, [*context*, ...])

predicate: (Function, Rule, or Data Type Constructor) Expression that returns a Boolean (true or false).

list: (Any Type Array) List of elements that the predicate iterates through.

context: (Any Type Array) Variable number of parameters passed directly into each predicate evaluation.

Returns

Boolean

Notes

Use `fn!functionName` to reference an expression function and `rule!ruleName` to reference a rule.

Returns false as soon as the returned value of an evaluation yields true; otherwise, returns true.

Empty and null lists yield true.

Serves as an alternative to `not(or(apply(rule!iseven, {1, 2, 3})))` and `not(any(rule!iseven, {-1,0,1,2}))` .

Examples

Copy and paste an example into the Test Rules interface to see how this works.

Given a function `f(x)` , `none(fn!f, {a, b, c}, v)` returns `not(or({f(a, v), f(b, v), f(c, v)}))`

`none(rule!iseven,{-1,0,1,2})` returns `false`

Returns a Single Value or a New Array

The following Looping Functions call a rule or function for each item in a list and returns a subset of that list, reduces the list down to a single value, or creates a new list based on modifications made to values in the list.

filter()

Calls a predicate for each item in a list and returns any items for which the returned value is true.

Syntax

filter(*predicate*, *list*, [*context*, ...])

predicate: (Function, Rule, or Data Type Constructor) Expression that returns a Boolean (true or false).

list: (Any Type Array) List of elements that the predicate iterates through.

context: (Any Type Array) Variable number of parameters passed directly into each predicate evaluation.

Returns

Any Type Array

Notes

Use `fn!functionName` to reference an expression function and `rule!ruleName` to reference a rule.

Examples

Copy and paste an example into the Test Rules interface to see how this works.

`filter(fn!iseven,{-1,0,1,2},1)` returns `0, 2`

reduce()

Calls a rule or function for each item in a list, passing the result of each call to the next one, and returns the value of the last computation.

Syntax

reduce(*function*, *initial*, *list*, [*context*,...])

function: (Rule or Function Reference) Rule or expression function.

initial: (Any Type) The accumulator's initial value.

list: (Any Type) Array of elements that the function iterates through.

context: (Any Type Array) Variable number of parameters passed directly into each function evaluation.

Returns

Any Type

Notes

Use `fn!functionName` to reference an expression function and `rule!ruleName` to reference a rule.

Best used when the computation of each operation needs to use the result from the previous operation such as when the result of each operation is an array that should be appended to each other in order. The result from the previous operation is then passed to the subsequent operation as the parameter initial.

The initial accumulator value is given by initial.

Null lists return a null list without executing the function.

Returns a scalar value if the function called returns a scalar, and returns a list if the function called returns a list.

To use rules or functions that take more than two arguments, use the merge() function.

- For example, given a rule `g(x, y, z)`, `reduce(rule!g, i, merge({a, b, c}, {d, e, f}))` returns `g(g(g(i, a, d), b, e), c, f)`.

Examples

Copy and paste an example into the Test Rules interface to see how this works.

Given a function `h(x, y)`, `reduce(fn!h,initial, {a, b, c}, v)` returns `h(h(h(initial, a, v), b, v), c, v)`.

`reduce(fn!sum, 0, {1, 2, 3})` returns `6`

apply()

Calls a rule or function for each item in a list, and provides any contexts specified.

Syntax

apply(*function*, *list*, [*context*,...])

function: (Rule or Function Reference) Rule or expression function.

list: (Any Type Array) List of elements that function iterates through.

context: (Any Type Array) Variable number of parameters passed directly into each function evaluation.

Returns

Any Type Array

Notes

Use `fn!functionName` to reference an expression function and `rule!ruleName` to reference a rule.

Null lists return a null list without executing the function.

To use rules or functions that take more than one argument, use the `merge()` function. For example, given a rule `g(x, y), apply(rule!g, merge({a, b, c}, {d, e, f}))` returns `{g(a, d), g(b, e), g(c, f)}`.

The result of each operation is appended to each other in the same order as their corresponding item in the input list. If the result of each operation is a array, `apply()` returns a two-dimensional array which can then be used for further computation. When the two-dimensional array is saved into a proce variable, a node input or a custom data type, the array is flattened to a one-dimensional array. Local variables created using the `with()` and `load()` functions, however, can store the two-dimensional array without flattening it.

See also: with() and load()

If you save the nested arrays into a process variable for multiple values, the nested function is flattened. Keep in mind that casting to a flattened array only happens when saving into a process variable, node input, custom data type, or custom data type field.

See also: Arrays in Expressions

To avoid having the nested function flatten, you can use the output of the `apply()` function as the input for a `merge()` function.

See below: merge()

`apply()` cannot be used with rules or functions that store local data, including rules using `load()` and certain SAIL components. In these cases, `apply()` will return an error. In these cases, use `a!applyComponents()`.

See also: load() and a!applyComponents().

Examples

Copy and paste an example into the Test Rules interface to see how this works.

Given a function `h(x, y)`, `apply(fn!h, {a, b, c}, v)` returns `{h(a, v), h(b, v) ,h(c, v)}`

`apply(fn!sum,{-1,2,3},2)` returns `1, 4, 5`

reject()

Calls a predicate for each item in a list, rejects any items for which the returned value is true, and returns all remaining items.

Syntax

reject(*predicate*, *list*, [*context*, ...])

predicate: (Function, Rule, or Data Type Constructor) Expression that returns a Boolean (true or false).

list: (Any Type Array) List of elements that the predicate iterates through.

context: (Any Type Array) Variable number of parameters passed directly into each function evaluation.

Returns

Any Type Array

Notes

Use `fn!functionName` to reference an expression function and `rule!ruleName` to reference a rule.

Examples

Copy and paste an example into the Test Rules interface to see how this works.

`reject(fn!isnull, {1, null(), 3})` returns `1, 3`

merge()

Takes a variable number of lists and merges them into a single list (or a list of lists) that is the size of the largest list provided.

Syntax

merge([*list*, ...])

list: (Any Type Array) Variable number of lists to merge into one list.

Returns

Any Type

Notes

Shorter lists are padded with null entries.

Use this function when you have a looping function referencing a rule or function that takes more than one argument. The order of the argument mus match the order of your rule input parameters.

Examples

Copy and paste an example into the Test Rules interface to see how this works.

`merge({1, 2, 3}, {4, 5, 6})` returns `1, 4, 2, 5, 3, 6`

User Guides by Role

- Designer
- Developer
- Web Admin
- Server Admin (On-Premise Only)

Tutorials

- Records
- Interfaces
- Process

Release Information

- Release Notes
- Installation
- Migration

System Requirements

Hotfixes

Release History

Other

STAR Methodology

Best Practices

Glossary

APIs