Search (S)

Overview
Tutorial
Reference

LANGUAGE
Syntax
Styling
Scripting
Context

LIBRARY
Foundations
Arguments
Array
Assert
Auto
Boolean
Bytes
Calculation
Content
Datetime
Dictionary
Duration
Evaluate
Float
Function
Integer
Label
Module

Reference > Foundations > Array

array

A sequence of values.

You can construct an array by enclosing a comma-separated sequence of values in parentheses. The values do not have to be of the same type.

You can access and update array items with the `.at()` method. Indices are zero-based and negative indices wrap around to the end of the array. You can iterate over an array using a for loop. Arrays can be added together with the + operator, joined together and multiplied with integers.

Note: An array of length one needs a trailing comma, as in `(1,)`. This is to disambiguate from a simple parenthesized expressions like `(1 + 2) * 3`. An empty array is written as `()`.

Example

```
#let values = (1, 7, 4, -3, 2)

#values.at(0) \
#(values.at(0) = 3)
#values.at(-1) \
#values.find(calc.even) \
#values.filter(calc.odd) \
#values.map(calc.abs) \
#values.rev() \
#(1, (2, 3)).flatten() \
#(("A", "B", "C")
    .join(", ", last: " and "))
```

```
1
2
4
```

```
(3, 7, -3)
(3, 7, 4, 3, 2)
(2, -3, 4, 7, 3)
(1, 2, 3)
A, B and C
```

Constructor

Converts a value to an array.

Note that this function is only intended for conversion of a collection-like value to an array, not for creation of an array from individual items. Use the array syntax `(1, 2, 3)` (or `(1,)` for a single-element array) instead.

```
array( bytes  array  version ) -> array
```

```
#let hi = "Hello 😃"
#array(bytes(hi))
```

```
(72, 101, 108, 108, 111, 32, 240, 159,
152, 131)
```

value

`bytes` or `array` or `version` *Required* *Positional*

The value that should be converted to an array.

Definitions

len

The number of values in the array.

```
self.len() -> int
```

first

Returns the first item in the array. May be used on the

left-hand side of an assignment. Fails with an error if the array is empty.

```
self.first() -> any
```

last

Returns the last item in the array. May be used on the left-hand side of an assignment. Fails with an error if the array is empty.

```
self.last() -> any
```

at

Returns the item at the specified index in the array. May be used on the left-hand side of an assignment. Returns the default value if the index is out of bounds or fails with an error if no default value was specified.

```
self.at(
  int,
  default: any,
) -> any
```

index `int` *Required* *Positional*

The index at which to retrieve the item. If negative, indexes from the back.

default `any`

A default value to return if the index is out of bounds.

push

Adds a value to the end of the array.

```
self.push( any )
```

value `any` *Required* *Positional*

The value to insert at the end of the array.

pop

Removes the last item from the array and returns it. Fails with an error if the array is empty.

```
self.pop() -> any
```

insert

Inserts a value into the array at the specified index. Fails with an error if the index is out of bounds.

```
self.insert(
  int,
  any,
)
```

index `int` *Required* *Positional*

The index at which to insert the item. If negative, indexes from the back.

value `any` *Required* *Positional*

The value to insert into the array.

remove

Removes the value at the specified index from the array and return it.

```
self.remove(
  int,
  default: any,
) -> any
```

index `int` *Required* *Positional*

The index at which to remove the item. If negative, indexes from the back.

default `any`

A default value to return if the index is out of bounds.

slice

Extracts a subslice of the array. Fails with an error if the start or index is out of bounds.

```
self.slice(
  int,
  none int,
  count: int,
) -> array
```

start `int` *Required* *Positional*

The start index (inclusive). If negative, indexes from the back.

end `none` or `int` *Positional*

The end index (exclusive). If omitted, the whole slice until the end of the array is extracted. If negative, indexes from the back.

Default: `none`

count `int`

The number of items to extract. This is equivalent to passing `start + count` as the `end` position. Mutually exclusive with `end`.

contains

Whether the array contains the specified value.

This method also has dedicated syntax: You can write `2 in (1, 2, 3)` instead of `(1, 2, 3).contains(2)`.

```
self.contains( any ) -> bool
```

value `any` *Required* *Positional*

The value to search for.

find

Searches for an item for which the given function returns `true` and returns the first match or `none` if there is no match.

```
self.find( function ) -> any none
```

searcher `function` *Required* *Positional*

The function to apply to each item. Must return a boolean.

position

Searches for an item for which the given function returns `true` and returns the index of the first match or `none` if there is no match.

```
self.position( function ) -> none int
```

searcher `function` *Required* *Positional*

The function to apply to each item. Must return a boolean.

range

Create an array consisting of a sequence of numbers.

If you pass just one positional parameter, it is interpreted as the `end` of the range. If you pass two, they describe the `start` and `end` of the range.

This function is available both in the array function's scope and globally.

```
array.range(
  int,
```

```
  int,
  step: int,
) -> array
```

```
#range(5) \
#range(2, 5) \
#range(20, step: 4) \
#range(21, step: 4) \
#range(5, 2, step: -1)
```

```
(0, 1, 2, 3, 4)
(2, 3, 4)
(0, 4, 8, 12, 16)
(0, 4, 8, 12, 16, 20)
(5, 4, 3)
```

start `int` *Positional*

The start of the range (inclusive).

Default: `0`

end `int` *Required* *Positional*

The end of the range (exclusive).

step `int`

The distance between the generated numbers.

Default: `1`

filter

Produces a new array with only the items from the original one for which the given function returns true.

```
self.filter(function) -> array
```

test `function` *Required* *Positional*

The function to apply to each item. Must return a boolean.

map

Produces a new array in which all items from the original one were transformed with the given function.

```
self.map( function ) -> array
```

mapper `function` *Required* *Positional*

The function to apply to each item.

enumerate

Returns a new array with the values alongside their indices.

The returned array consists of `(index, value)` pairs in the form of length-2 arrays. These can be destructured with a let binding or for loop.

```
self.enumerate( start: int ) -> array
```

start `int`

The index returned for the first pair of the returned list.

Default: `0`

zip

Zips the array with other arrays.

Returns an array of arrays, where the `i`th inner array contains all the `i`th elements from each original array.

If the arrays to be zipped have different lengths, they are zipped up to the last element of the shortest array and all remaining elements are ignored.

This function is variadic, meaning that you can zip multiple arrays together at once: `(1, 2).zip(("A", "B"), (10, 20))` yields `((1, "A", 10), (2, "B", 20))`.

```
self.zip( .. array ) -> array
```

others `array` *Required* *Positional* *Variadic*

The arrays to zip with.

fold

Folds all items into a single value using an accumulator function.

```
self.fold(
  any,
  function,
) -> any
```

init `any` *Required* *Positional*

The initial value to start with.

folder `function` *Required* *Positional*

The folding function. Must have two parameters: One for the accumulated value and one for an item.

sum

Sums all items (works for all types that can be added).

```
self.sum(default: any) -> any
```

default `any`

What to return if the array is empty. Must be set if the array can be empty.

product

Calculates the product all items (works for all types that can be multiplied).

```
self.product(default: any) -> any
```

default `any`

What to return if the array is empty. Must be set if the array can be empty.

any

Whether the given function returns `true` for any item in the array.

```
self.any( function ) -> bool
```

test `function` *Required* *Positional*

The function to apply to each item. Must return a boolean.

all

Whether the given function returns `true` for all items in the array.

```
self.all( function ) -> bool
```

test `function` *Required* *Positional*

The function to apply to each item. Must return a boolean.

flatten

Combine all nested arrays into a single flat one.

```
self.flatten() -> array
```

rev

Return a new array with the same items, but in reverse order.

```
self.rev() -> array
```

split

Split the array at occurrences of the specified value.

```
self.split( any ) -> array
```

at `any` *Required* *Positional*

The value to split at.

join

Combine all items in the array into one.

```
self.join(
  any none,
  last: any,
) -> any
```

separator `any` or `none` *Positional*

A value to insert between each item of the array.

Default: none

last `any`

An alternative separator between the last two items.

intersperse

Returns an array with a copy of the separator value placed between adjacent elements.

```
self.intersperse( any ) -> array
```

separator `any` *Required* *Positional*

The value that will be placed between each adjacent element.

chunks

Splits an array into non-overlapping chunks, starting at the beginning, ending with a single remainder chunk.

All chunks but the last have `chunk-size` elements. If `exact` is set to `true`, the remainder is dropped if it contains less than `chunk-size` elements.

```
self.chunks(
  int,
  exact: bool,
) -> array
```

```
#let array = (1, 2, 3, 4, 5, 6, 7, 8)
#array.chunks(3)
#array.chunks(3, exact: true)
```

```
((1, 2, 3), (4, 5, 6), (7, 8)) ((1, 2, 3), (4, 5, 6))
```

`chunk-size` `int` *Required* *Positional*

How many elements each chunk may at most contain.

`exact` `bool`

Whether to keep the remainder if its size is less than `chunk-size`.

Default: `false`

sorted

Return a sorted version of this array, optionally by a given key function. The sorting algorithm used is stable.

Returns an error if two values could not be compared or if the key function (if given) yields an error.

```
self.sorted(key: function) -> array
```

key `function`

If given, applies this function to the elements in the array to determine the keys to sort by.

dedup

Deduplicates all items in the array.

Returns a new array with all duplicate items removed. Only the first element of each duplicate is kept.

```
self.dedup(key: function) -> array
```

```
#(1, 1, 2, 3, 1).dedup()
```

```
(1, 2, 3)
```

key `function`

If given, applies this function to the elements in the array to determine the keys to deduplicate by.

Arguments
Previous page

Assert
Next page