“Array” Methods

`_.chunk(array, [size=1])`

Creates an array of elements split into groups the length of `size`. If `array` can't be split evenly, the final chunk will be the remaining elements.

Since

3.0.0

Arguments

`array` ***(Array)***: The array to process.
`[size=1]` ***(number)***: The length of each chunk

Returns

(Array): Returns the new array of chunks.

Example

```
_.chunk(['a', 'b', 'c', 'd'], 2);
// => [['a', 'b'], ['c', 'd']]

_.chunk(['a', 'b', 'c', 'd'], 3);
// => [['a', 'b', 'c'], ['d']]
```

`_.compact(array)`

Creates an array with all falsey values removed. The values `false`, `null`, `0`, `""`, `undefined`, and `NaN` are falsey.

Since

0.1.0

Arguments

`array` ***(Array)***: The array to compact.

Returns

(Array): Returns the new array of filtered values.

Example

```
_.compact([0, 1, false, 2, '', 3]);
// => [1, 2, 3]
```

_.concat(array, [values])

Creates a new array concatenating `array` with any additional arrays and/or values.

Since

4.0.0

Arguments

array ***(Array)***: The array to concatenate.
[values] ***(...*)***: The values to concatenate.

Returns

(Array): Returns the new concatenated array.

Example

```
var array = [1];
var other = _.concat(array, 2, [3], [[4]]);

console.log(other);
// => [1, 2, 3, [4]]

console.log(array);
// => [1]
```

_.difference(array, [values])

Creates an array of `array` values not included in the other given arrays using `SameValueZero` (http://ecma-international.org/ecma-262/7.0/#sec-samevaluezero) for equality comparisons. The order and references of result values are determined by the first array.

Note: Unlike `_.pullAll`, this method returns a new array.

Since

0.1.0

Arguments

array ***(Array)***: The array to inspect.
[values] ***(...Array)***: The values to exclude.

Returns

(Array): Returns the new array of filtered values.

Example

```
_.difference([2, 1], [2, 3]);
// => [1]
```

`_.differenceBy(array, [values], [iteratee=_.identity])`

This method is like `_.difference` except that it accepts `iteratee` which is invoked for each element of `array` and `values` to generate the criterion by which they're compared. The order and references of result values are determined by the first array. The iteratee is invoked with one argument:
(value).

Note: Unlike `_.pullAllBy`, this method returns a new array.

Since

4.0.0

Arguments

array ***(Array)***: The array to inspect.
[values] ***(...Array)***: The values to exclude.
[iteratee=_.identity] ***(Function)***: The iteratee invoked per element.

Returns

(Array): Returns the new array of filtered values.

Example

```
_.differenceBy([2.1, 1.2], [2.3, 3.4], Math.floor);
// => [1.2]

// The `_.property` iteratee shorthand.
_.differenceBy([{ 'x': 2 }, { 'x': 1 }], [{ 'x': 1 }], 'x');
// => [{ 'x': 2 }]
```

_.differenceWith(array, [values], [comparator])

This method is like `_.difference` except that it accepts `comparator` which is invoked to compare elements of `array` to `values`. The order and references of result values are determined by the first array. The comparator is invoked with two arguments: *(arrVal, othVal)*.

Note: Unlike `_.pullAllWith`, this method returns a new array.

Since

4.0.0

Arguments

array ***(Array)***: The array to inspect.
[values] ***(...Array)***: The values to exclude.
[comparator] ***(Function)***: The comparator invoked per element.

Returns

(Array): Returns the new array of filtered values.

Example

```
var objects = [{ 'x': 1, 'y': 2 }, { 'x': 2, 'y': 1 }];

_.differenceWith(objects, [{ 'x': 1, 'y': 2 }], _.isEqual);
// => [{ 'x': 2, 'y': 1 }]
```

_.drop(array, [n=1])

Creates a slice of `array` with `n` elements dropped from the beginning.

Since

0.5.0

Arguments

array ***(Array)***: The array to query.
[n=1] ***(number)***: The number of elements to drop.

Returns

(Array): Returns the slice of `array`.

Example

```
_.drop([1, 2, 3]);
// => [2, 3]

_.drop([1, 2, 3], 2);
// => [3]

_.drop([1, 2, 3], 5);
// => []

_.drop([1, 2, 3], 0);
// => [1, 2, 3]
```

_.dropRight(array, [n=1])

Creates a slice of `array` with `n` elements dropped from the end.

Since

3.0.0

Arguments

`array` ***(Array)***: The array to query.
`[n=1]` ***(number)***: The number of elements to drop.

Returns

(Array): Returns the slice of `array`.

Example

```
_.dropRight([1, 2, 3]);
// => [1, 2]

_.dropRight([1, 2, 3], 2);
// => [1]

_.dropRight([1, 2, 3], 5);
// => []

_.dropRight([1, 2, 3], 0);
// => [1, 2, 3]
```

.dropRightWhile(array, [predicate=.identity])

Creates a slice of `array` excluding elements dropped from the end. Elements are dropped until `predicate` returns falsey. The predicate is invoked with three arguments: *(value, index, array)*.

Since

3.0.0

Arguments

array ***(Array)***: The array to query.
[predicate=_.identity] ***(Function)***: The function invoked per iteration.

Returns

(Array): Returns the slice of `array`.

Example

```
var users = [
  { 'user': 'barney',  'active': true },
  { 'user': 'fred',    'active': false },
  { 'user': 'pebbles', 'active': false }
];

_.dropRightWhile(users, function(o) { return !o.active; });
// => objects for ['barney']

// The `_.matches` iteratee shorthand.
_.dropRightWhile(users, { 'user': 'pebbles', 'active': false });
// => objects for ['barney', 'fred']

// The `_.matchesProperty` iteratee shorthand.
_.dropRightWhile(users, ['active', false]);
// => objects for ['barney']

// The `_.property` iteratee shorthand.
_.dropRightWhile(users, 'active');
// => objects for ['barney', 'fred', 'pebbles']
```

.dropWhile(array, [predicate=.identity])

Creates a slice of `array` excluding elements dropped from the beginning. Elements are dropped until `predicate` returns falsey. The predicate is invoked with three arguments: *(value, index, array)*.

Since

3.0.0

Arguments

`array` ***(Array)***: The array to query.
`[predicate=_.identity]` ***(Function)***: The function invoked per iteration.

Returns

(Array): Returns the slice of `array`.

Example

```
var users = [
  { 'user': 'barney',  'active': false },
  { 'user': 'fred',    'active': false },
  { 'user': 'pebbles', 'active': true }
];

_.dropWhile(users, function(o) { return !o.active; });
// => objects for ['pebbles']

// The `_.matches` iteratee shorthand.
_.dropWhile(users, { 'user': 'barney', 'active': false });
// => objects for ['fred', 'pebbles']

// The `_.matchesProperty` iteratee shorthand.
_.dropWhile(users, ['active', false]);
// => objects for ['pebbles']

// The `_.property` iteratee shorthand.
_.dropWhile(users, 'active');
// => objects for ['barney', 'fred', 'pebbles']
```

`_.fill(array, value, [start=0], [end=array.length])`

Fills elements of `array` with `value` from `start` up to, but not including, `end`.

Note: This method mutates `array`.

Since

3.2.0

Arguments

`array` ***(Array)***: The array to fill.
`value` ***(*)***: The value to fill **`array`** with.
`[start=0]` ***(number)***: The start position.
`[end=array.length]` ***(number)***: The end position.

Returns

(Array): Returns `array`.

Example

```
var array = [1, 2, 3];

_.fill(array, 'a');
console.log(array);
// => ['a', 'a', 'a']

_.fill(Array(3), 2);
// => [2, 2, 2]

_.fill([4, 6, 8, 10], '*', 1, 3);
// => [4, '*', '*', 10]
```

`_.findIndex(array, [predicate=_.identity], [fromIndex=0])`

This method is like `_.find` except that it returns the index of the first element `predicate` returns truthy for instead of the element itself.

Since

1.1.0

Arguments

`array` ***(Array)***: The array to inspect.
`[predicate=_.identity]` ***(Function)***: The function invoked per iteration.
`[fromIndex=0]` ***(number)***: The index to search from.

Returns

(number): Returns the index of the found element, else `-1`.

Example

```
var users = [
  { 'user': 'barney',  'active': false },
  { 'user': 'fred',    'active': false },
  { 'user': 'pebbles', 'active': true }
];

_.findIndex(users, function(o) { return o.user == 'barney'; });
// => 0

// The `_.matches` iteratee shorthand.
_.findIndex(users, { 'user': 'fred', 'active': false });
// => 1

// The `_.matchesProperty` iteratee shorthand.
_.findIndex(users, ['active', false]);
// => 0

// The `_.property` iteratee shorthand.
_.findIndex(users, 'active');
// => 2
```

.findLastIndex(array, [predicate=.identity], [fromIndex=array.length-1])

This method is like `_.findIndex` except that it iterates over elements of `collection` from right to left.

Since

2.0.0

Arguments

array ***(Array)***: The array to inspect.
[predicate=_.identity] ***(Function)***: The function invoked per iteration.
[fromIndex=array.length-1] ***(number)***: The index to search from.

Returns

(number): Returns the index of the found element, else `-1`.

Example

```
var users = [
  { 'user': 'barney',  'active': true },
  { 'user': 'fred',    'active': false },
  { 'user': 'pebbles', 'active': false }
];

_.findLastIndex(users, function(o) { return o.user == 'pebbles'; });
// => 2

// The `_.matches` iteratee shorthand.
_.findLastIndex(users, { 'user': 'barney', 'active': true });
// => 0

// The `_.matchesProperty` iteratee shorthand.
_.findLastIndex(users, ['active', false]);
// => 2

// The `_.property` iteratee shorthand.
_.findLastIndex(users, 'active');
// => 0
```

_.flatten(array)

Flattens `array` a single level deep.

Since

0.1.0

Arguments

array ***(Array)***: The array to flatten.

Returns

(Array): Returns the new flattened array.

Example

```
_.flatten([1, [2, [3, [4]], 5]]);
// => [1, 2, [3, [4]], 5]
```

_.flattenDeep(array)

Recursively flattens `array`.

Since

3.0.0

Arguments

`array` ***(Array)***: The array to flatten.

Returns

(Array): Returns the new flattened array.

Example

```
_.flattenDeep([1, [2, [3, [4]], 5]]);
// => [1, 2, 3, 4, 5]
```

`_.flattenDepth(array, [depth=1])`

Recursively flatten `array` up to `depth` times.

Since

4.4.0

Arguments

`array` ***(Array)***: The array to flatten.
`[depth=1]` ***(number)***: The maximum recursion depth.

Returns

(Array): Returns the new flattened array.

Example

```
var array = [1, [2, [3, [4]], 5]];

_.flattenDepth(array, 1);
// => [1, 2, [3, [4]], 5]

_.flattenDepth(array, 2);
// => [1, 2, 3, [4], 5]
```

`_.fromPairs(pairs)`

The inverse of `_.toPairs`; this method returns an object composed from key-value `pairs`.

Since

4.0.0

Arguments

`pairs` ***(Array)***: The key-value pairs.

Returns

(Object): Returns the new object.

Example

```
_.fromPairs([['a', 1], ['b', 2]]);
// => { 'a': 1, 'b': 2 }
```

_.head(array)

Gets the first element of `array`.

Since

0.1.0

Aliases

_.first

Arguments

`array` ***(Array)***: The array to query.

Returns

(*): Returns the first element of `array`.

Example

```
_.head([1, 2, 3]);
// => 1

_.head([]);
// => undefined
```

`_.indexOf(array, value, [fromIndex=0])`

Gets the index at which the first occurrence of `value` is found in `array` using `SameValueZero` (http://ecma-international.org/ecma-262/7.0/#sec-samevaluezero) for equality comparisons. If `fromIndex` is negative, it's used as the offset from the end of `array`.

Since

0.1.0

Arguments

`array` ***(Array)***: The array to inspect.
`value` ***(*)***: The value to search for.
`[fromIndex=0]` ***(number)***: The index to search from.

Returns

(number): Returns the index of the matched value, else `-1`.

Example

```
_.indexOf([1, 2, 1, 2], 2);
// => 1

// Search from the `fromIndex`.
_.indexOf([1, 2, 1, 2], 2, 2);
// => 3
```

`_.initial(array)`

Gets all but the last element of `array`.

Since

0.1.0

Arguments

`array` ***(Array)***: The array to query.

Returns

(Array): Returns the slice of `array`.

Example

```
_.initial([1, 2, 3]);
// => [1, 2]
```

_.intersection([arrays])

Creates an array of unique values that are included in all given arrays using `SameValueZero` (http://ecma-international.org/ecma-262/7.0/#sec-samevaluezero) for equality comparisons. The order and references of result values are determined by the first array.

Since

0.1.0

Arguments

[arrays] ***(...Array)***: The arrays to inspect.

Returns

(Array): Returns the new array of intersecting values.

Example

```
_.intersection([2, 1], [2, 3]);
// => [2]
```

.intersectionBy([arrays], [iteratee=.identity])

This method is like `_.intersection` except that it accepts `iteratee` which is invoked for each element of each `arrays` to generate the criterion by which they're compared. The order and references of result values are determined by the first array. The iteratee is invoked with one argument:
(value).

Since

4.0.0

Arguments

[arrays] ***(...Array)***: The arrays to inspect.
[iteratee=_.identity] ***(Function)***: The iteratee invoked per element.

Returns

(Array): Returns the new array of intersecting values.

Example

```
_.intersectionBy([2.1, 1.2], [2.3, 3.4], Math.floor);
// => [2.1]

// The `_.property` iteratee shorthand.
_.intersectionBy([{ 'x': 1 }], [{ 'x': 2 }, { 'x': 1 }], 'x');
// => [{ 'x': 1 }]
```

_.intersectionWith([arrays], [comparator])

This method is like `_.intersection` except that it accepts `comparator` which is invoked to compare elements of `arrays`. The order and references of result values are determined by the first array. The comparator is invoked with two arguments: *(arrVal, othVal)*.

Since

4.0.0

Arguments

[arrays] ***(...Array)***: The arrays to inspect.
[comparator] ***(Function)***: The comparator invoked per element.

Returns

(Array): Returns the new array of intersecting values.

Example

```
var objects = [{ 'x': 1, 'y': 2 }, { 'x': 2, 'y': 1 }];
var others = [{ 'x': 1, 'y': 1 }, { 'x': 1, 'y': 2 }];

_.intersectionWith(objects, others, _.isEqual);
// => [{ 'x': 1, 'y': 2 }]
```

_.join(array, [separator=','])

Converts all elements in `array` into a string separated by `separator`.

Since

4.0.0

Arguments

array ***(Array)***: The array to convert.
[separator=','] ***(string)***: The element separator.

Returns

(string): Returns the joined string.

Example

```
_.join(['a', 'b', 'c'], '~');
// => 'a~b~c'
```

_.last(array)

Gets the last element of `array`.

Since

0.1.0

Arguments

array ***(Array)***: The array to query.

Returns

(*): Returns the last element of `array`.

Example

```
_.last([1, 2, 3]);
// => 3
```

_.lastIndexOf(array, value, [fromIndex=array.length-1])

This method is like `_.indexOf` except that it iterates over elements of `array` from right to left.

Since

0.1.0

Arguments

array ***(Array)***: The array to inspect.
value ***(*)***: The value to search for.
[fromIndex=array.length-1] ***(number)***: The index to search from.

Returns

(number): Returns the index of the matched value, else `-1`.

Example

```
_.lastIndexOf([1, 2, 1, 2], 2);
// => 3

// Search from the `fromIndex`.
_.lastIndexOf([1, 2, 1, 2], 2, 2);
// => 1
```

_.nth(array, [n=0])

Gets the element at index n of `array`. If n is negative, the nth element from the end is returned.

Since

4.11.0

Arguments

array *(Array)*: The array to query.
[n=0] *(number)*: The index of the element to return.

Returns

(*): Returns the nth element of `array`.

Example

```
var array = ['a', 'b', 'c', 'd'];

_.nth(array, 1);
// => 'b'

_.nth(array, -2);
// => 'c';
```

_.pull(array, [values])

Removes all given values from `array` using `SameValueZero` (http://ecma-international.org/ecma-262/7.0/#sec-samevaluezero) for equality comparisons.

Note: Unlike `_.without`, this method mutates `array`. Use `_.remove` to remove elements from an array by predicate.

Since

2.0.0

Arguments

`array` ***(Array)***: The array to modify.
`[values]` ***(...*)***: The values to remove.

Returns

(Array): Returns `array`.

Example

```
var array = ['a', 'b', 'c', 'a', 'b', 'c'];

_.pull(array, 'a', 'c');
console.log(array);
// => ['b', 'b']
```

`_.pullAll(array, values)`

This method is like `_.pull` except that it accepts an array of values to remove.

Note: Unlike `_.difference`, this method mutates `array`.

Since

4.0.0

Arguments

`array` ***(Array)***: The array to modify.
`values` ***(Array)***: The values to remove.

Returns

(Array): Returns `array`.

Example

```
var array = ['a', 'b', 'c', 'a', 'b', 'c'];

_.pullAll(array, ['a', 'c']);
console.log(array);
// => ['b', 'b']
```

.pullAllBy(array, values, [iteratee=.identity])

This method is like `_.pullAll` except that it accepts `iteratee` which is invoked for each element of `array` and `values` to generate the criterion by which they're compared. The iteratee is invoked with one argument: *(value)*.

Note: Unlike `_.differenceBy`, this method mutates `array`.

Since

4.0.0

Arguments

`array` *(Array)*: The array to modify.
`values` *(Array)*: The values to remove.
`[iteratee=_.identity]` *(Function)*: The iteratee invoked per element.

Returns

(Array): Returns `array`.

Example

```
var array = [{ 'x': 1 }, { 'x': 2 }, { 'x': 3 }, { 'x': 1 }];

_.pullAllBy(array, [{ 'x': 1 }, { 'x': 3 }], 'x');
console.log(array);
// => [{ 'x': 2 }]
```

_.pullAllWith(array, values, [comparator])

This method is like `_.pullAll` except that it accepts `comparator` which is invoked to compare elements of `array` to `values`. The comparator is invoked with two arguments: *(arrVal, othVal)*.

Note: Unlike `_.differenceWith`, this method mutates `array`.

Since

4.6.0

Arguments

array ***(Array)***: The array to modify.
values ***(Array)***: The values to remove.
[comparator] ***(Function)***: The comparator invoked per element.

Returns

(Array): Returns `array`.

Example

```
var array = [{ 'x': 1, 'y': 2 }, { 'x': 3, 'y': 4 }, { 'x': 5, 'y': 6 }];

_.pullAllWith(array, [{ 'x': 3, 'y': 4 }], _.isEqual);
console.log(array);
// => [{ 'x': 1, 'y': 2 }, { 'x': 5, 'y': 6 }]
```

`_.pullAt(array, [indexes])`

Removes elements from `array` corresponding to `indexes` and returns an array of removed elements.

Note: Unlike `_.at`, this method mutates `array`.

Since

3.0.0

Arguments

array ***(Array)***: The array to modify.
[indexes] ***(...(number|number[]))***: The indexes of elements to remove.

Returns

(Array): Returns the new array of removed elements.

Example

```
var array = ['a', 'b', 'c', 'd'];
var pulled = _.pullAt(array, [1, 3]);

console.log(array);
// => ['a', 'c']

console.log(pulled);
// => ['b', 'd']
```

.remove(array, [predicate=.identity])

Removes all elements from `array` that `predicate` returns truthy for and returns an array of the removed elements. The predicate is invoked with three arguments: *(value, index, array)*.

Note: Unlike `_.filter`, this method mutates `array`. Use `_.pull` to pull elements from an array by value.

Since

2.0.0

Arguments

`array` ***(Array)***: The array to modify.
`[predicate=_.identity]` ***(Function)***: The function invoked per iteration.

Returns

(Array): Returns the new array of removed elements.

Example

```
var array = [1, 2, 3, 4];
var evens = _.remove(array, function(n) {
  return n % 2 == 0;
});

console.log(array);
// => [1, 3]

console.log(evens);
// => [2, 4]
```

_.reverse(array)

Reverses `array` so that the first element becomes the last, the second element becomes the second to last, and so on.

Note: This method mutates `array` and is based on `Array#reverse` (https://mdn.io/Array/reverse).

Since

4.0.0

Arguments

`array` ***(Array)***: The array to modify.

Returns

(Array): Returns `array`.

Example

```
var array = [1, 2, 3];

_.reverse(array);
// => [3, 2, 1]

console.log(array);
// => [3, 2, 1]
```

`_.slice(array, [start=0], [end=array.length])`

Creates a slice of `array` from `start` up to, but not including, `end`.

Note: This method is used instead of `Array#slice` (https://mdn.io/Array/slice) to ensure dense arrays are returned.

Since

3.0.0

Arguments

`array` ***(Array)***: The array to slice.
`[start=0]` ***(number)***: The start position.
`[end=array.length]` ***(number)***: The end position.

Returns

(Array): Returns the slice of `array`.

`_.sortedIndex(array, value)`

Uses a binary search to determine the lowest index at which `value` should be inserted into `array` in order to maintain its sort order.

Since

0.1.0

Arguments

`array` ***(Array)***: The sorted array to inspect.
`value` ***(*)***: The value to evaluate.

Returns

(number): Returns the index at which `value` should be inserted into `array`.

Example

```
_.sortedIndex([30, 50], 40);
// => 1
```

`_.sortedIndexBy(array, value, [iteratee=_.identity])`

This method is like `_.sortedIndex` except that it accepts `iteratee` which is invoked for `value` and each element of `array` to compute their sort ranking. The iteratee is invoked with one argument: *(value)*.

Since

4.0.0

Arguments

`array` ***(Array)***: The sorted array to inspect.
`value` ***(*)***: The value to evaluate.
`[iteratee=_.identity]` ***(Function)***: The iteratee invoked per element.

Returns

(number): Returns the index at which `value` should be inserted into `array`.

Example

```
var objects = [{ 'x': 4 }, { 'x': 5 }];

_.sortedIndexBy(objects, { 'x': 4 }, function(o) { return o.x; });
// => 0

// The `_.property` iteratee shorthand.
_.sortedIndexBy(objects, { 'x': 4 }, 'x');
// => 0
```

_.sortedIndexOf(array, value)

This method is like `_.indexOf` except that it performs a binary search on a sorted `array`.

Since

4.0.0

Arguments

`array` ***(Array)***: The array to inspect.
`value` ***(*)***: The value to search for.

Returns

(number): Returns the index of the matched value, else `-1`.

Example

```
_.sortedIndexOf([4, 5, 5, 5, 6], 5);
// => 1
```

_.sortedLastIndex(array, value)

This method is like `_.sortedIndex` except that it returns the highest index at which `value` should be inserted into `array` in order to maintain its sort order.

Since

3.0.0

Arguments

`array` ***(Array)***: The sorted array to inspect.
`value` ***(*)***: The value to evaluate.

Returns

(number): Returns the index at which `value` should be inserted into `array`.

Example

```
_.sortedLastIndex([4, 5, 5, 5, 6], 5);
// => 4
```

.sortedLastIndexBy(array, value, [iteratee=.identity])

This method is like `_.sortedLastIndex` except that it accepts `iteratee` which is invoked for `value` and each element of `array` to compute their sort ranking. The iteratee is invoked with one argument: *(value)*.

Since

4.0.0

Arguments

array ***(Array)***: The sorted array to inspect.
value ***(*)***: The value to evaluate.
[iteratee=_.identity] ***(Function)***: The iteratee invoked per element.

Returns

(number): Returns the index at which `value` should be inserted into `array`.

Example

```
var objects = [{ 'x': 4 }, { 'x': 5 }];

_.sortedLastIndexBy(objects, { 'x': 4 }, function(o) { return o.x; });
// => 1

// The `_.property` iteratee shorthand.
_.sortedLastIndexBy(objects, { 'x': 4 }, 'x');
// => 1
```

_.sortedLastIndexOf(array, value)

This method is like `_.lastIndexOf` except that it performs a binary search on a sorted `array`.

Since

4.0.0

Arguments

array ***(Array)***: The array to inspect.
value ***(*)***: The value to search for.

Returns

(number): Returns the index of the matched value, else `-1`.

Example

```
_.sortedLastIndexOf([4, 5, 5, 5, 6], 5);
// => 3
```

_.sortedUniq(array)

This method is like `_.uniq` except that it's designed and optimized for sorted arrays.

Since

4.0.0

Arguments

array ***(Array)***: The array to inspect.

Returns

(Array): Returns the new duplicate free array.

Example

```
_.sortedUniq([1, 1, 2]);
// => [1, 2]
```

_.sortedUniqBy(array, [iteratee])

This method is like `_.uniqBy` except that it's designed and optimized for sorted arrays.

Since

4.0.0

Arguments

array ***(Array)***: The array to inspect.
[iteratee] ***(Function)***: The iteratee invoked per element.

Returns

(Array): Returns the new duplicate free array.

Example

```
_.sortedUniqBy([1.1, 1.2, 2.3, 2.4], Math.floor);
// => [1.1, 2.3]
```

_.tail(array)

Gets all but the first element of `array`.

Since

4.0.0

Arguments

array ***(Array)***: The array to query.

Returns

(Array): Returns the slice of `array`.

Example

```
_.tail([1, 2, 3]);
// => [2, 3]
```

_.take(array, [n=1])

Creates a slice of `array` with `n` elements taken from the beginning.

Since

0.1.0

Arguments

array ***(Array)***: The array to query.
[n=1] ***(number)***: The number of elements to take.

Returns

(Array): Returns the slice of `array`.

Example

```
_.take([1, 2, 3]);
// => [1]

_.take([1, 2, 3], 2);
// => [1, 2]

_.take([1, 2, 3], 5);
// => [1, 2, 3]

_.take([1, 2, 3], 0);
// => []
```

_.takeRight(array, [n=1])

Creates a slice of `array` with `n` elements taken from the end.

Since

3.0.0

Arguments

array ***(Array)***: The array to query.
[n=1] ***(number)***: The number of elements to take.

Returns

(Array): Returns the slice of `array`.

Example

```
_.takeRight([1, 2, 3]);
// => [3]

_.takeRight([1, 2, 3], 2);
// => [2, 3]

_.takeRight([1, 2, 3], 5);
// => [1, 2, 3]

_.takeRight([1, 2, 3], 0);
// => []
```

.takeRightWhile(array, [predicate=.identity])

Creates a slice of `array` with elements taken from the end. Elements are taken until `predicate` returns falsey. The predicate is invoked with three arguments: *(value, index, array)*.

Since

3.0.0

Arguments

`array` ***(Array)***: The array to query.
`[predicate=_.identity]` ***(Function)***: The function invoked per iteration.

Returns

(Array): Returns the slice of `array`.

Example

```
var users = [
  { 'user': 'barney',  'active': true },
  { 'user': 'fred',    'active': false },
  { 'user': 'pebbles', 'active': false }
];

_.takeRightWhile(users, function(o) { return !o.active; });
// => objects for ['fred', 'pebbles']

// The `_.matches` iteratee shorthand.
_.takeRightWhile(users, { 'user': 'pebbles', 'active': false });
// => objects for ['pebbles']

// The `_.matchesProperty` iteratee shorthand.
_.takeRightWhile(users, ['active', false]);
// => objects for ['fred', 'pebbles']

// The `_.property` iteratee shorthand.
_.takeRightWhile(users, 'active');
// => []
```

`_.takeWhile(array, [predicate=_.identity])`

Creates a slice of `array` with elements taken from the beginning. Elements are taken until `predicate` returns falsey. The predicate is invoked with three arguments: *(value, index, array)*.

Since

3.0.0

Arguments

array ***(Array)***: The array to query.
[predicate=_.identity] ***(Function)***: The function invoked per iteration.

Returns

(Array): Returns the slice of array.

Example

```
var users = [
  { 'user': 'barney',  'active': false },
  { 'user': 'fred',    'active': false },
  { 'user': 'pebbles', 'active': true }
];

_.takeWhile(users, function(o) { return !o.active; });
// => objects for ['barney', 'fred']

// The `_.matches` iteratee shorthand.
_.takeWhile(users, { 'user': 'barney', 'active': false });
// => objects for ['barney']

// The `_.matchesProperty` iteratee shorthand.
_.takeWhile(users, ['active', false]);
// => objects for ['barney', 'fred']

// The `_.property` iteratee shorthand.
_.takeWhile(users, 'active');
// => []
```

_.union([arrays])

Creates an array of unique values, in order, from all given arrays using `SameValueZero` (http://ecma-international.org/ecma-262/7.0/#sec-samevaluezero) for equality comparisons.

Since

0.1.0

Arguments

[arrays] ***(...Array)***: The arrays to inspect.

Returns

(Array): Returns the new array of combined values.

Example

```
_.union([2], [1, 2]);
// => [2, 1]
```

.unionBy([arrays], [iteratee=.identity])

This method is like `_.union` except that it accepts `iteratee` which is invoked for each element of each `arrays` to generate the criterion by which uniqueness is computed. Result values are chosen from the first array in which the value occurs. The iteratee is invoked with one argument:
(value).

Since

4.0.0

Arguments

[arrays] ***(...Array)***: The arrays to inspect.
[iteratee=_.identity] ***(Function)***: The iteratee invoked per element.

Returns

(Array): Returns the new array of combined values.

Example

```
_.unionBy([2.1], [1.2, 2.3], Math.floor);
// => [2.1, 1.2]

// The `_.property` iteratee shorthand.
_.unionBy([{ 'x': 1 }], [{ 'x': 2 }, { 'x': 1 }], 'x');
// => [{ 'x': 1 }, { 'x': 2 }]
```

_.unionWith([arrays], [comparator])

This method is like `_.union` except that it accepts `comparator` which is invoked to compare elements of `arrays`. Result values are chosen from the first array in which the value occurs. The comparator is invoked with two arguments: *(arrVal, othVal)*.

Since

4.0.0

Arguments

[arrays] ***(...Array)***: The arrays to inspect.
[comparator] ***(Function)***: The comparator invoked per element.

Returns

(Array): Returns the new array of combined values.

Example

```
var objects = [{ 'x': 1, 'y': 2 }, { 'x': 2, 'y': 1 }];
var others = [{ 'x': 1, 'y': 1 }, { 'x': 1, 'y': 2 }];

_.unionWith(objects, others, _.isEqual);
// => [{ 'x': 1, 'y': 2 }, { 'x': 2, 'y': 1 }, { 'x': 1, 'y': 1 }]
```

`_.uniq(array)`

Creates a duplicate-free version of an array, using `SameValueZero` (http://ecma-international.org/ecma-262/7.0/#sec-samevaluezero) for equality comparisons, in which only the first occurrence of each element is kept. The order of result values is determined by the order they occur in the array.

Since

0.1.0

Arguments

array ***(Array)***: The array to inspect.

Returns

(Array): Returns the new duplicate free array.

Example

```
_.uniq([2, 1, 2]);
// => [2, 1]
```

`_.uniqBy(array, [iteratee=_.identity])`

This method is like `_.uniq` except that it accepts `iteratee` which is invoked for each element in `array` to generate the criterion by which uniqueness is computed. The order of result values is determined by the order they occur in the array. The iteratee is invoked with one argument:
(value).

Since

4.0.0

Arguments

array ***(Array)***: The array to inspect.
[iteratee=_.identity] ***(Function)***: The iteratee invoked per element.

Returns

(Array): Returns the new duplicate free array.

Example

```
_.uniqBy([2.1, 1.2, 2.3], Math.floor);
// => [2.1, 1.2]

// The `_.property` iteratee shorthand.
_.uniqBy([{ 'x': 1 }, { 'x': 2 }, { 'x': 1 }], 'x');
// => [{ 'x': 1 }, { 'x': 2 }]
```

_.uniqWith(array, [comparator])

This method is like `_.uniq` except that it accepts `comparator` which is invoked to compare elements of `array`. The order of result values is determined by the order they occur in the array.The comparator is invoked with two arguments: *(arrVal, othVal)*.

Since

4.0.0

Arguments

array ***(Array)***: The array to inspect.
[comparator] ***(Function)***: The comparator invoked per element.

Returns

(Array): Returns the new duplicate free array.

Example

```
var objects = [{ 'x': 1, 'y': 2 }, { 'x': 2, 'y': 1 }, { 'x': 1, 'y': 2 }];

_.uniqWith(objects, _.isEqual);
// => [{ 'x': 1, 'y': 2 }, { 'x': 2, 'y': 1 }]
```

`_.unzip(array)`

This method is like `_.zip` except that it accepts an array of grouped elements and creates an array regrouping the elements to their pre-zip configuration.

Since

1.2.0

Arguments

`array` ***(Array)***: The array of grouped elements to process.

Returns

(Array): Returns the new array of regrouped elements.

Example

```
var zipped = _.zip(['a', 'b'], [1, 2], [true, false]);
// => [['a', 1, true], ['b', 2, false]]

_.unzip(zipped);
// => [['a', 'b'], [1, 2], [true, false]]
```

`_.unzipWith(array, [iteratee=_.identity])`

This method is like `_.unzip` except that it accepts `iteratee` to specify how regrouped values should be combined. The iteratee is invoked with the elements of each group: *(...group)*.

Since

3.8.0

Arguments

`array` ***(Array)***: The array of grouped elements to process.
`[iteratee=_.identity]` ***(Function)***: The function to combine regrouped values.

Returns

(Array): Returns the new array of regrouped elements.

Example

```
var zipped = _.zip([1, 2], [10, 20], [100, 200]);
// => [[1, 10, 100], [2, 20, 200]]

_.unzipWith(zipped, _.add);
// => [3, 30, 300]
```

`_.without(array, [values])`

Creates an array excluding all given values using `SameValueZero` (http://ecma-international.org/ecma-262/7.0/#sec-samevaluezero) for equality comparisons.

Note: Unlike `_.pull`, this method returns a new array.

Since

0.1.0

Arguments

array *(Array)*: The array to inspect.
[values] *(...*)*: The values to exclude.

Returns

(Array): Returns the new array of filtered values.

Example

```
_.without([2, 1, 2, 3], 1, 2);
// => [3]
```

`_.xor([arrays])`

Creates an array of unique values that is the symmetric difference (https://en.wikipedia.org/wiki/Symmetric_difference) of the given arrays. The order of result values is determined by the order they occur in the arrays.

Since

2.4.0

Arguments

[arrays] *(...Array)*: The arrays to inspect.

Returns

(Array): Returns the new array of filtered values.

Example

```
_.xor([2, 1], [2, 3]);
// => [1, 3]
```

.xorBy([arrays], [iteratee=.identity])

This method is like `_.xor` except that it accepts `iteratee` which is invoked for each element of each `arrays` to generate the criterion by which by which they're compared. The order of result values is determined by the order they occur in the arrays. The iteratee is invoked with one argument: *(value)*.

Since

4.0.0

Arguments

[arrays] ***(...Array)***: The arrays to inspect.
[iteratee=_.identity] ***(Function)***: The iteratee invoked per element.

Returns

(Array): Returns the new array of filtered values.

Example

```
_.xorBy([2.1, 1.2], [2.3, 3.4], Math.floor);
// => [1.2, 3.4]

// The `_.property` iteratee shorthand.
_.xorBy([{ 'x': 1 }], [{ 'x': 2 }, { 'x': 1 }], 'x');
// => [{ 'x': 2 }]
```

_.xorWith([arrays], [comparator])

This method is like `_.xor` except that it accepts `comparator` which is invoked to compare elements of `arrays`. The order of result values is determined by the order they occur in the arrays. The comparator is invoked with two arguments: *(arrVal, othVal)*.

Since

4.0.0

Arguments

[arrays] ***(...Array)***: The arrays to inspect.
[comparator] ***(Function)***: The comparator invoked per element.

Returns

(Array): Returns the new array of filtered values.

Example

```
var objects = [{ 'x': 1, 'y': 2 }, { 'x': 2, 'y': 1 }];
var others = [{ 'x': 1, 'y': 1 }, { 'x': 1, 'y': 2 }];

_.xorWith(objects, others, _.isEqual);
// => [{ 'x': 2, 'y': 1 }, { 'x': 1, 'y': 1 }]
```

_.zip([arrays])

Creates an array of grouped elements, the first of which contains the first elements of the given arrays, the second of which contains the second elements of the given arrays, and so on.

Since

0.1.0

Arguments

[arrays] ***(...Array)***: The arrays to process.

Returns

(Array): Returns the new array of grouped elements.

Example

```
_.zip(['a', 'b'], [1, 2], [true, false]);
// => [['a', 1, true], ['b', 2, false]]
```

_.zipObject([props=[]], [values=[]])

This method is like `_.fromPairs` except that it accepts two arrays, one of property identifiers and one of corresponding values.

Since

0.4.0

Arguments

[props=[]] ***(Array)***: The property identifiers.
[values=[]] ***(Array)***: The property values.

Returns

(Object): Returns the new object.

Example

```
_.zipObject(['a', 'b'], [1, 2]);
// => { 'a': 1, 'b': 2 }
```

`_.zipObjectDeep([props=[]], [values=[]])`

This method is like `_.zipObject` except that it supports property paths.

Since

4.1.0

Arguments

[props=[]] ***(Array)***: The property identifiers.
[values=[]] ***(Array)***: The property values.

Returns

(Object): Returns the new object.

Example

```
_.zipObjectDeep(['a.b[0].c', 'a.b[1].d'], [1, 2]);
// => { 'a': { 'b': [{ 'c': 1 }, { 'd': 2 }] } }
```

`_.zipWith([arrays], [iteratee=_.identity])`

This method is like `_.zip` except that it accepts `iteratee` to specify how grouped values should be combined. The iteratee is invoked with the elements of each group: *(...group)*.

Since

3.8.0

Arguments

[arrays] ***(...Array)***: The arrays to process.
[iteratee=_.identity] ***(Function)***: The function to combine grouped values.

Returns

(Array): Returns the new array of grouped elements.

Example

```
_.zipWith([1, 2], [10, 20], [100, 200], function(a, b, c) {
  return a + b + c;
});
// => [111, 222]
```

“Collection” Methods

.countBy(collection, [iteratee=.identity])

Creates an object composed of keys generated from the results of running each element of `collection` thru `iteratee`. The corresponding value of each key is the number of times the key was returned by `iteratee`. The iteratee is invoked with one argument: *(value)*.

Since

0.5.0

Arguments

collection ***(Array|Object)***: The collection to iterate over.
[iteratee=_.identity] ***(Function)***: The iteratee to transform keys.

Returns

(Object): Returns the composed aggregate object.

Example

```
_.countBy([6.1, 4.2, 6.3], Math.floor);
// => { '4': 1, '6': 2 }

// The `_.property` iteratee shorthand.
_.countBy(['one', 'two', 'three'], 'length');
// => { '3': 2, '5': 1 }
```

.every(collection, [predicate=.identity])

Checks if `predicate` returns truthy for **all** elements of `collection`. Iteration is stopped once `predicate` returns falsey. The predicate is invoked with three arguments: *(value, index|key, collection)*.

Note: This method returns `true` for empty collections (https://en.wikipedia.org/wiki/Empty_set) because everything is true (https://en.wikipedia.org/wiki/Vacuous_truth) of elements of empty collections.

Since

0.1.0

Arguments

`collection` ***(Array|Object)***: The collection to iterate over.
`[predicate=_.identity]` ***(Function)***: The function invoked per iteration.

Returns

(boolean): Returns `true` if all elements pass the predicate check, else `false`.

Example

```
_.every([true, 1, null, 'yes'], Boolean);
// => false

var users = [
  { 'user': 'barney', 'age': 36, 'active': false },
  { 'user': 'fred',   'age': 40, 'active': false }
];

// The `_.matches` iteratee shorthand.
_.every(users, { 'user': 'barney', 'active': false });
// => false

// The `_.matchesProperty` iteratee shorthand.
_.every(users, ['active', false]);
// => true

// The `_.property` iteratee shorthand.
_.every(users, 'active');
// => false
```

`_.filter(collection, [predicate=_.identity])`

Iterates over elements of `collection`, returning an array of all elements `predicate` returns truthy for. The predicate is invoked with three arguments: *(value, index|key, collection)*.

Note: Unlike `_.remove`, this method returns a new array.

Since

0.1.0

Arguments

`collection` ***(Array|Object)***: The collection to iterate over.
`[predicate=_.identity]` ***(Function)***: The function invoked per iteration.

Returns

(Array): Returns the new filtered array.

Example

```
var users = [
  { 'user': 'barney', 'age': 36, 'active': true },
  { 'user': 'fred',   'age': 40, 'active': false }
];

_.filter(users, function(o) { return !o.active; });
// => objects for ['fred']

// The `_.matches` iteratee shorthand.
_.filter(users, { 'age': 36, 'active': true });
// => objects for ['barney']

// The `_.matchesProperty` iteratee shorthand.
_.filter(users, ['active', false]);
// => objects for ['fred']

// The `_.property` iteratee shorthand.
_.filter(users, 'active');
// => objects for ['barney']
```

`_.find(collection, [predicate=_.identity], [fromIndex=0])`

Iterates over elements of `collection`, returning the first element `predicate` returns truthy for. The predicate is invoked with three arguments: *(value, index|key, collection)*.

Since

0.1.0

Arguments

collection ***(Array|Object)***: The collection to inspect.
[predicate=_.identity] ***(Function)***: The function invoked per iteration.
[fromIndex=0] ***(number)***: The index to search from.

Returns

(*): Returns the matched element, else `undefined`.

Example

```
var users = [
  { 'user': 'barney',  'age': 36, 'active': true },
  { 'user': 'fred',    'age': 40, 'active': false },
  { 'user': 'pebbles', 'age': 1,  'active': true }
];

_.find(users, function(o) { return o.age < 40; });
// => object for 'barney'

// The `_.matches` iteratee shorthand.
_.find(users, { 'age': 1, 'active': true });
// => object for 'pebbles'

// The `_.matchesProperty` iteratee shorthand.
_.find(users, ['active', false]);
// => object for 'fred'

// The `_.property` iteratee shorthand.
_.find(users, 'active');
// => object for 'barney'
```

.findLast(collection, [predicate=.identity], [fromIndex=collection.lengt

This method is like `_.find` except that it iterates over elements of `collection` from right to left.

Since

2.0.0

Arguments

collection ***(Array|Object)***: The collection to inspect.
[predicate=_.identity] ***(Function)***: The function invoked per iteration.
[fromIndex=collection.length-1] ***(number)***: The index to search from.

Returns

(*): Returns the matched element, else `undefined`.

Example

```
_.findLast([1, 2, 3, 4], function(n) {
  return n % 2 == 1;
});
// => 3
```

.flatMap(collection, [iteratee=.identity])

Creates a flattened array of values by running each element in `collection` thru `iteratee` and flattening the mapped results. The iteratee is invoked with three arguments: *(value, index|key, collection)*.

Since

4.0.0

Arguments

`collection` ***(Array|Object)***: The collection to iterate over.
`[iteratee=_.identity]` ***(Function)***: The function invoked per iteration.

Returns

(Array): Returns the new flattened array.

Example

```
function duplicate(n) {
  return [n, n];
}

_.flatMap([1, 2], duplicate);
// => [1, 1, 2, 2]
```

.flatMapDeep(collection, [iteratee=.identity])

This method is like `_.flatMap` except that it recursively flattens the mapped results.

Since

4.7.0

Arguments

collection ***(Array|Object)***: The collection to iterate over.
[iteratee=_.identity] ***(Function)***: The function invoked per iteration.

Returns

(Array): Returns the new flattened array.

Example

```
function duplicate(n) {
  return [[[n, n]]];
}

_.flatMapDeep([1, 2], duplicate);
// => [1, 1, 2, 2]
```

.flatMapDepth(collection, [iteratee=.identity], [depth=1])

This method is like `_.flatMap` except that it recursively flattens the mapped results up to `depth` times.

Since

4.7.0

Arguments

collection ***(Array|Object)***: The collection to iterate over.
[iteratee=_.identity] ***(Function)***: The function invoked per iteration.
[depth=1] ***(number)***: The maximum recursion depth.

Returns

(Array): Returns the new flattened array.

Example

```
function duplicate(n) {
  return [[[n, n]]];
}

_.flatMapDepth([1, 2], duplicate, 2);
// => [[1, 1], [2, 2]]
```

.forEach(collection, [iteratee=.identity])

Iterates over elements of `collection` and invokes `iteratee` for each element. The iteratee is invoked with three arguments: *(value, index|key, collection)*. Iteratee functions may exit iteration early by explicitly returning `false`.

Note: As with other "Collections" methods, objects with a "length" property are iterated like arrays. To avoid this behavior use `_.forIn` or `_.forOwn` for object iteration.

Since

0.1.0

Aliases

_.each

Arguments

`collection` ***(Array|Object)***: The collection to iterate over.
`[iteratee=_.identity]` ***(Function)***: The function invoked per iteration.

Returns

(*): Returns `collection`.

Example

```
_.forEach([1, 2], function(value) {
  console.log(value);
});
// => Logs `1` then `2`.

_.forEach({ 'a': 1, 'b': 2 }, function(value, key) {
  console.log(key);
});
// => Logs 'a' then 'b' (iteration order is not guaranteed).
```

`_.forEachRight(collection, [iteratee=_.identity])`

This method is like `_.forEach` except that it iterates over elements of `collection` from right to left.

Since

2.0.0

Aliases

_.eachRight

Arguments

`collection` ***(Array|Object)***: The collection to iterate over.
`[iteratee=_.identity]` ***(Function)***: The function invoked per iteration.

Returns

(*): Returns `collection`.

Example

```
_.forEachRight([1, 2], function(value) {
  console.log(value);
});
// => Logs `2` then `1`.
```

`_.groupBy(collection, [iteratee=_.identity])`

Creates an object composed of keys generated from the results of running each element of `collection` thru `iteratee`. The order of grouped values is determined by the order they occur in `collection`. The corresponding value of each key is an array of elements responsible for generating the key. The iteratee is invoked with one argument: *(value)*.

Since

0.1.0

Arguments

`collection` ***(Array|Object)***: The collection to iterate over.
`[iteratee=_.identity]` ***(Function)***: The iteratee to transform keys.

Returns

(Object): Returns the composed aggregate object.

Example

```
_.groupBy([6.1, 4.2, 6.3], Math.floor);
// => { '4': [4.2], '6': [6.1, 6.3] }

// The `_.property` iteratee shorthand.
_.groupBy(['one', 'two', 'three'], 'length');
// => { '3': ['one', 'two'], '5': ['three'] }
```

`_.includes(collection, value, [fromIndex=0])`

Checks if `value` is in `collection`. If `collection` is a string, it's checked for a substring of `value`, otherwise `SameValueZero` (http://ecma-international.org/ecma-262/7.0/#sec-samevaluezero) is used for equality comparisons. If `fromIndex` is negative, it's used as the offset from the end of `collection`.

Since

0.1.0

Arguments

`collection` ***(Array|Object|string)***: The collection to inspect.
`value` ***(*)***: The value to search for.
`[fromIndex=0]` ***(number)***: The index to search from.

Returns

(boolean): Returns `true` if `value` is found, else `false`.

Example

```
_.includes([1, 2, 3], 1);
// => true

_.includes([1, 2, 3], 1, 2);
// => false

_.includes({ 'a': 1, 'b': 2 }, 1);
// => true

_.includes('abcd', 'bc');
// => true
```

_.invokeMap(collection, path, [args])

Invokes the method at `path` of each element in `collection`, returning an array of the results of each invoked method. Any additional arguments are provided to each invoked method. If `path` is a function, it's invoked for, and `this` bound to, each element in `collection`.

Since

4.0.0

Arguments

`collection` ***(Array|Object)***: The collection to iterate over.

path ***(Array|Function|string)***: The path of the method to invoke or the function invoked per iteration.
[args] ***(...*)***: The arguments to invoke each method with.

Returns

(Array): Returns the array of results.

Example

```
_.invokeMap([[5, 1, 7], [3, 2, 1]], 'sort');
// => [[1, 5, 7], [1, 2, 3]]

_.invokeMap([123, 456], String.prototype.split, '');
// => [['1', '2', '3'], ['4', '5', '6']]
```

`_.keyBy(collection, [iteratee=_.identity])`

Creates an object composed of keys generated from the results of running each element of `collection` thru `iteratee`. The corresponding value of each key is the last element responsible for generating the key. The iteratee is invoked with one argument: *(value)*.

Since

4.0.0

Arguments

collection ***(Array|Object)***: The collection to iterate over.
[iteratee=_.identity] ***(Function)***: The iteratee to transform keys.

Returns

(Object): Returns the composed aggregate object.

Example

```
var array = [
  { 'dir': 'left', 'code': 97 },
  { 'dir': 'right', 'code': 100 }
];

_.keyBy(array, function(o) {
  return String.fromCharCode(o.code);
});
// => { 'a': { 'dir': 'left', 'code': 97 }, 'd': { 'dir': 'right', 'code': 100 } }

_.keyBy(array, 'dir');
// => { 'left': { 'dir': 'left', 'code': 97 }, 'right': { 'dir': 'right', 'code': 100
```

.map(collection, [iteratee=.identity])

Creates an array of values by running each element in `collection` thru `iteratee`. The iteratee is invoked with three arguments:
(value, index|key, collection).

Many lodash methods are guarded to work as iteratees for methods like `_.every`, `_.filter`, `_.map`, `_.mapValues`, `_.reject`, and `_.some`.

The guarded methods are:
`ary`, `chunk`, `curry`, `curryRight`, `drop`, `dropRight`, `every`, `fill`, `invert`, `parseInt`, `random`, `range`, `rangeRight`, `repeat`, `sampleSize`, `slice`, `some`, `sortBy`, `split`, `take`, `takeRight`, `template`, `trim`, `trimEnd`, `trimStart`, and `words`

Since

0.1.0

Arguments

`collection` ***(Array|Object)***: The collection to iterate over.
`[iteratee=_.identity]` ***(Function)***: The function invoked per iteration.

Returns

(Array): Returns the new mapped array.

Example

```
function square(n) {
  return n * n;
}

_.map([4, 8], square);
// => [16, 64]

_.map({ 'a': 4, 'b': 8 }, square);
// => [16, 64] (iteration order is not guaranteed)

var users = [
  { 'user': 'barney' },
  { 'user': 'fred' }
];

// The `_.property` iteratee shorthand.
_.map(users, 'user');
// => ['barney', 'fred']
```

.orderBy(collection, [iteratees=[.identity]], [orders])

This method is like `_.sortBy` except that it allows specifying the sort orders of the iteratees to sort by. If `orders` is unspecified, all values are sorted in ascending order. Otherwise, specify an order of "desc" for descending or "asc" for ascending sort order of corresponding values.

Since

4.0.0

Arguments

collection *(Array|Object)*: The collection to iterate over.
[iteratees=[_.identity]] *(Array[]|Function[]|Object[]|string[])*: The iteratees to sort by.
[orders] *(string[])*: The sort orders of **iteratees**.

Returns

(Array): Returns the new sorted array.

Example

```
var users = [
  { 'user': 'fred',   'age': 48 },
  { 'user': 'barney', 'age': 34 },
  { 'user': 'fred',   'age': 40 },
  { 'user': 'barney', 'age': 36 }
];

// Sort by `user` in ascending order and by `age` in descending order.
_.orderBy(users, ['user', 'age'], ['asc', 'desc']);
// => objects for [['barney', 36], ['barney', 34], ['fred', 48], ['fred', 40]]
```

.partition(collection, [predicate=.identity])

Creates an array of elements split into two groups, the first of which contains elements `predicate` returns truthy for, the second of which contains elements `predicate` returns falsey for. The predicate is invoked with one argument: *(value)*.

Since

3.0.0

Arguments

`collection` ***(Array|Object)***: The collection to iterate over.
`[predicate=_.identity]` ***(Function)***: The function invoked per iteration.

Returns

(Array): Returns the array of grouped elements.

Example

```
var users = [
  { 'user': 'barney',  'age': 36, 'active': false },
  { 'user': 'fred',    'age': 40, 'active': true },
  { 'user': 'pebbles', 'age': 1,  'active': false }
];

_.partition(users, function(o) { return o.active; });
// => objects for [['fred'], ['barney', 'pebbles']]

// The `_.matches` iteratee shorthand.
_.partition(users, { 'age': 1, 'active': false });
// => objects for [['pebbles'], ['barney', 'fred']]

// The `_.matchesProperty` iteratee shorthand.
_.partition(users, ['active', false]);
// => objects for [['barney', 'pebbles'], ['fred']]

// The `_.property` iteratee shorthand.
_.partition(users, 'active');
// => objects for [['fred'], ['barney', 'pebbles']]
```

.reduce(collection, [iteratee=.identity], [accumulator])

Reduces `collection` to a value which is the accumulated result of running each element in `collection` thru `iteratee`, where each successive invocation is supplied the return value of the previous. If `accumulator` is not given, the first element of `collection` is used as the initial value. The iteratee is invoked with four arguments:
(accumulator, value, index|key, collection).

Many lodash methods are guarded to work as iteratees for methods like `_.reduce`, `_.reduceRight`, and `_.transform`.

The guarded methods are:
`assign`, `defaults`, `defaultsDeep`, `includes`, `merge`, `orderBy`, and `sortBy`

Since

0.1.0

Arguments

`collection` ***(Array|Object)***: The collection to iterate over.
`[iteratee=_.identity]` ***(Function)***: The function invoked per iteration.
`[accumulator]` ***(*)***: The initial value.

Returns

(*): Returns the accumulated value.

Example

```
_.reduce([1, 2], function(sum, n) {
  return sum + n;
}, 0);
// => 3

_.reduce({ 'a': 1, 'b': 2, 'c': 1 }, function(result, value, key) {
  (result[value] || (result[value] = [])).push(key);
  return result;
}, {});
// => { '1': ['a', 'c'], '2': ['b'] } (iteration order is not guaranteed)
```

.reduceRight(collection, [iteratee=.identity], [accumulator])

This method is like `_.reduce` except that it iterates over elements of `collection` from right to left.

Since

0.1.0

Arguments

collection *(Array|Object)*: The collection to iterate over.
[iteratee=_.identity] *(Function)*: The function invoked per iteration.
[accumulator] *(*)*: The initial value.

Returns

(*): Returns the accumulated value.

Example

```
var array = [[0, 1], [2, 3], [4, 5]];

_.reduceRight(array, function(flattened, other) {
  return flattened.concat(other);
}, []);
// => [4, 5, 2, 3, 0, 1]
```

.reject(collection, [predicate=.identity])

The opposite of `_.filter`; this method returns the elements of `collection` that `predicate` does **not** return truthy for.

Since

0.1.0

Arguments

`collection` ***(Array|Object)***: The collection to iterate over.
`[predicate=_.identity]` ***(Function)***: The function invoked per iteration.

Returns

(Array): Returns the new filtered array.

Example

```
var users = [
  { 'user': 'barney', 'age': 36, 'active': false },
  { 'user': 'fred',   'age': 40, 'active': true }
];

_.reject(users, function(o) { return !o.active; });
// => objects for ['fred']

// The `_.matches` iteratee shorthand.
_.reject(users, { 'age': 40, 'active': true });
// => objects for ['barney']

// The `_.matchesProperty` iteratee shorthand.
_.reject(users, ['active', false]);
// => objects for ['fred']

// The `_.property` iteratee shorthand.
_.reject(users, 'active');
// => objects for ['barney']
```

`_.sample(collection)`

Gets a random element from `collection`.

Since

2.0.0

Arguments

`collection` *(Array|Object)*: The collection to sample.

Returns

(*): Returns the random element.

Example

```
_.sample([1, 2, 3, 4]);
// => 2
```

`_.sampleSize(collection, [n=1])`

Gets n random elements at unique keys from `collection` up to the size of `collection`.

Since

4.0.0

Arguments

`collection` *(Array|Object)*: The collection to sample.
`[n=1]` *(number)*: The number of elements to sample.

Returns

(Array): Returns the random elements.

Example

```
_.sampleSize([1, 2, 3], 2);
// => [3, 1]

_.sampleSize([1, 2, 3], 4);
// => [2, 3, 1]
```

`_.shuffle(collection)`

Creates an array of shuffled values, using a version of the Fisher-Yates shuffle (https://en.wikipedia.org/wiki/Fisher-Yates_shuffle).

Since

0.1.0

Arguments

`collection` ***`(Array|Object)`***: The collection to shuffle.

Returns

`(Array)`: Returns the new shuffled array.

Example

```
_.shuffle([1, 2, 3, 4]);
// => [4, 1, 3, 2]
```

`_.size(collection)`

Gets the size of `collection` by returning its length for array-like values or the number of own enumerable string keyed properties for objects.

Since

0.1.0

Arguments

`collection` ***`(Array|Object|string)`***: The collection to inspect.

Returns

`(number)`: Returns the collection size.

Example

```
_.size([1, 2, 3]);
// => 3

_.size({ 'a': 1, 'b': 2 });
// => 2

_.size('pebbles');
// => 7
```

`_.some(collection, [predicate=_.identity])`

Checks if `predicate` returns truthy for **any** element of `collection`. Iteration is stopped once `predicate` returns truthy. The predicate is invoked with three arguments: *(value, index|key, collection)*.

Since

0.1.0

Arguments

`collection` ***(Array|Object)***: The collection to iterate over.
`[predicate=_.identity]` ***(Function)***: The function invoked per iteration.

Returns

(boolean): Returns `true` if any element passes the predicate check, else `false`.

Example

```
_.some([null, 0, 'yes', false], Boolean);
// => true

var users = [
  { 'user': 'barney', 'active': true },
  { 'user': 'fred',   'active': false }
];

// The `_.matches` iteratee shorthand.
_.some(users, { 'user': 'barney', 'active': false });
// => false

// The `_.matchesProperty` iteratee shorthand.
_.some(users, ['active', false]);
// => true

// The `_.property` iteratee shorthand.
_.some(users, 'active');
// => true
```

`_.sortBy(collection, [iteratees=[_.identity]])`

Creates an array of elements, sorted in ascending order by the results of running each element in a collection thru each iteratee. This method performs a stable sort, that is, it preserves the original sort order of equal elements. The iteratees are invoked with one argument: *(value)*.

Since

0.1.0

Arguments

`collection` ***(Array|Object)***: The collection to iterate over.
`[iteratees=[_.identity]]` ***(...(Function|Function[]))***: The iteratees to sort by.

Returns

(Array): Returns the new sorted array.

Example

```
var users = [
  { 'user': 'fred',   'age': 48 },
  { 'user': 'barney', 'age': 36 },
  { 'user': 'fred',   'age': 40 },
  { 'user': 'barney', 'age': 34 }
];

_.sortBy(users, [function(o) { return o.user; }]);
// => objects for [['barney', 36], ['barney', 34], ['fred', 48], ['fred', 40]]

_.sortBy(users, ['user', 'age']);
// => objects for [['barney', 34], ['barney', 36], ['fred', 40], ['fred', 48]]
```

“Date” Methods

_.now()

Gets the timestamp of the number of milliseconds that have elapsed since the Unix epoch *(1 January 1970 00:00:00 UTC)*.

Since

2.4.0

Returns

(number): Returns the timestamp.

Example

```
_.defer(function(stamp) {
  console.log(_.now() - stamp);
}, _.now());
// => Logs the number of milliseconds it took for the deferred invocation.
```

“Function” Methods

_.after(n, func)

The opposite of `_.before`; this method creates a function that invokes `func` once it's called `n` or more times.

Since

0.1.0

Arguments

`n` *(number)*: The number of calls before **`func`** is invoked.
`func` *(Function)*: The function to restrict.

Returns

(Function): Returns the new restricted function.

Example

```
var saves = ['profile', 'settings'];

var done = _.after(saves.length, function() {
  console.log('done saving!');
});

_.forEach(saves, function(type) {
  asyncSave({ 'type': type, 'complete': done });
});
// => Logs 'done saving!' after the two async saves have completed.
```

`_.ary(func, [n=func.length])`

Creates a function that invokes `func`, with up to `n` arguments, ignoring any additional arguments.

Since

3.0.0

Arguments

`func` *(Function)*: The function to cap arguments for.
`[n=func.length]` *(number)*: The arity cap.

Returns

(Function): Returns the new capped function.

Example

```
_.map(['6', '8', '10'], _.ary(parseInt, 1));
// => [6, 8, 10]
```

_.before(n, func)

Creates a function that invokes `func`, with the `this` binding and arguments of the created function, while it's called less than `n` times. Subsequent calls to the created function return the result of the last `func` invocation.

Since

3.0.0

Arguments

`n` *(number)*: The number of calls at which **`func`** is no longer invoked.
`func` *(Function)*: The function to restrict.

Returns

(Function): Returns the new restricted function.

Example

```
jQuery(element).on('click', _.before(5, addContactToList));
// => Allows adding up to 4 contacts to the list.
```

_.bind(func, thisArg, [partials])

Creates a function that invokes `func` with the `this` binding of `thisArg` and `partials` prepended to the arguments it receives.

The `_.bind.placeholder` value, which defaults to _ in monolithic builds, may be used as a placeholder for partially applied arguments.

Note: Unlike native `Function#bind`, this method doesn't set the "length" property of bound functions.

Since

0.1.0

Arguments

`func` *(Function)*: The function to bind.
`thisArg` *(*)*: The **`this`** binding of **`func`**.

`[partials]` *`(...*)`*: The arguments to be partially applied.

Returns

`(Function)`: Returns the new bound function.

Example

```
function greet(greeting, punctuation) {
  return greeting + ' ' + this.user + punctuation;
}

var object = { 'user': 'fred' };

var bound = _.bind(greet, object, 'hi');
bound('!');
// => 'hi fred!'

// Bound with placeholders.
var bound = _.bind(greet, object, _, '!');
bound('hi');
// => 'hi fred!'
```

`_.bindKey(object, key, [partials])`

Creates a function that invokes the method at `object[key]` with `partials` prepended to the arguments it receives.

This method differs from `_.bind` by allowing bound functions to reference methods that may be redefined or don't yet exist. See Peter Michaux's article (http://peter.michaux.ca/articles/lazy-function-definition-pattern) for more details.

The `_.bindKey.placeholder` value, which defaults to `_` in monolithic builds, may be used as a placeholder for partially applied arguments.

Since

0.10.0

Arguments

`object` *`(Object)`*: The object to invoke the method on.
`key` *`(string)`*: The key of the method.
`[partials]` *`(...*)`*: The arguments to be partially applied.

Returns

(Function): Returns the new bound function.

Example

```
var object = {
  'user': 'fred',
  'greet': function(greeting, punctuation) {
    return greeting + ' ' + this.user + punctuation;
  }
};

var bound = _.bindKey(object, 'greet', 'hi');
bound('!');
// => 'hi fred!'

object.greet = function(greeting, punctuation) {
  return greeting + 'ya ' + this.user + punctuation;
};

bound('!');
// => 'hiya fred!'

// Bound with placeholders.
var bound = _.bindKey(object, 'greet', _, '!');
bound('hi');
// => 'hiya fred!'
```

`_.curry(func, [arity=func.length])`

Creates a function that accepts arguments of `func` and either invokes `func` returning its result, if at least `arity` number of arguments have been provided, or returns a function that accepts the remaining `func` arguments, and so on. The arity of `func` may be specified if `func.length` is not sufficient.

The `_.curry.placeholder` value, which defaults to _ in monolithic builds, may be used as a placeholder for provided arguments.

Note: This method doesn't set the "length" property of curried functions.

Since

2.0.0

Arguments

`func` ***(Function)***: The function to curry.
`[arity=func.length]` ***(number)***: The arity of **`func`**.

Returns

(Function): Returns the new curried function.

Example

```
var abc = function(a, b, c) {
  return [a, b, c];
};

var curried = _.curry(abc);

curried(1)(2)(3);
// => [1, 2, 3]

curried(1, 2)(3);
// => [1, 2, 3]

curried(1, 2, 3);
// => [1, 2, 3]

// Curried with placeholders.
curried(1)(_, 3)(2);
// => [1, 2, 3]
```

`_.curryRight(func, [arity=func.length])`

This method is like `_.curry` except that arguments are applied to `func` in the manner of `_.partialRight` instead of `_.partial`.

The `_.curryRight.placeholder` value, which defaults to `_` in monolithic builds, may be used as a placeholder for provided arguments.

Note: This method doesn't set the "length" property of curried functions.

Since

3.0.0

Arguments

`func` ***(Function)***: The function to curry.
`[arity=func.length]` ***(number)***: The arity of **`func`**.

Returns

(Function): Returns the new curried function.

Example

```
var abc = function(a, b, c) {
  return [a, b, c];
};

var curried = _.curryRight(abc);

curried(3)(2)(1);
// => [1, 2, 3]

curried(2, 3)(1);
// => [1, 2, 3]

curried(1, 2, 3);
// => [1, 2, 3]

// Curried with placeholders.
curried(3)(1, _)(2);
// => [1, 2, 3]
```

`_.debounce(func, [wait=0], [options={}])`

Creates a debounced function that delays invoking `func` until after `wait` milliseconds have elapsed since the last time the debounced function was invoked. The debounced function comes with a `cancel` method to cancel delayed `func` invocations and a `flush` method to immediately invoke them. Provide `options` to indicate whether `func` should be invoked on the leading and/or trailing edge of the `wait` timeout. The `func` is invoked with the last arguments provided to the debounced function. Subsequent calls to the debounced function return the result of the last `func` invocation.

Note: If `leading` and `trailing` options are `true`, `func` is invoked on the trailing edge of the timeout only if the debounced function is invoked more than once during the `wait` timeout.

If `wait` is `0` and `leading` is `false`, `func` invocation is deferred until to the next tick, similar to `setTimeout` with a timeout of `0`.

See David Corbacho's article (https://css-tricks.com/debouncing-throttling-explained-examples/) for details over the differences between `_.debounce` and `_.throttle`.

Since

0.1.0

Arguments

`func` ***(Function)***: The function to debounce.
`[wait=0]` ***(number)***: The number of milliseconds to delay.
`[options={}]` ***(Object)***: The options object.
`[options.leading=false]` ***(boolean)***: Specify invoking on the leading edge of the timeout.
`[options.maxWait]` ***(number)***: The maximum time **`func`** is allowed to be delayed before it's invoked.
`[options.trailing=true]` ***(boolean)***: Specify invoking on the trailing edge of the timeout.

Returns

(Function): Returns the new debounced function.

Example

```
// Avoid costly calculations while the window size is in flux.
jQuery(window).on('resize', _.debounce(calculateLayout, 150));

// Invoke `sendMail` when clicked, debouncing subsequent calls.
jQuery(element).on('click', _.debounce(sendMail, 300, {
  'leading': true,
  'trailing': false
}));

// Ensure `batchLog` is invoked once after 1 second of debounced calls.
var debounced = _.debounce(batchLog, 250, { 'maxWait': 1000 });
var source = new EventSource('/stream');
jQuery(source).on('message', debounced);

// Cancel the trailing debounced invocation.
jQuery(window).on('popstate', debounced.cancel);
```

`_.defer(func, [args])`

Defers invoking the `func` until the current call stack has cleared. Any additional arguments are provided to `func` when it's invoked.

Since

0.1.0

Arguments

`func` ***(Function)***: The function to defer.
`[args]` ***(...*)***: The arguments to invoke **`func`** with.

Returns

(number): Returns the timer id.

Example

```
_.defer(function(text) {
  console.log(text);
}, 'deferred');
// => Logs 'deferred' after one millisecond.
```

_.delay(func, wait, [args])

Invokes `func` after `wait` milliseconds. Any additional arguments are provided to `func` when it's invoked.

Since

0.1.0

Arguments

`func` ***(Function)***: The function to delay.
`wait` ***(number)***: The number of milliseconds to delay invocation.
`[args]` ***(...*)***: The arguments to invoke **`func`** with.

Returns

(number): Returns the timer id.

Example

```
_.delay(function(text) {
  console.log(text);
}, 1000, 'later');
// => Logs 'later' after one second.
```

_.flip(func)

Creates a function that invokes `func` with arguments reversed.

Since

4.0.0

Arguments

`func` ***(Function)***: The function to flip arguments for.

Returns

(Function): Returns the new flipped function.

Example

```
var flipped = _.flip(function() {
  return _.toArray(arguments);
});

flipped('a', 'b', 'c', 'd');
// => ['d', 'c', 'b', 'a']
```

`_.memoize(func, [resolver])`

Creates a function that memoizes the result of `func`. If `resolver` is provided, it determines the cache key for storing the result based on the arguments provided to the memoized function. By default, the first argument provided to the memoized function is used as the map cache key. The `func` is invoked with the `this` binding of the memoized function.

Note: The cache is exposed as the `cache` property on the memoized function. Its creation may be customized by replacing the `_.memoize.Cache` constructor with one whose instances implement the `Map` (http://ecma-international.org/ecma-262/7.0/#sec-properties-of-the-map-prototype-object) method interface of `clear`, `delete`, `get`, `has`, and `set`.

Since

0.1.0

Arguments

`func` ***(Function)***: The function to have its output memoized.
`[resolver]` ***(Function)***: The function to resolve the cache key.

Returns

(Function): Returns the new memoized function.

Example

```
var object = { 'a': 1, 'b': 2 };
var other = { 'c': 3, 'd': 4 };

var values = _.memoize(_.values);
values(object);
// => [1, 2]

values(other);
// => [3, 4]

object.a = 2;
values(object);
// => [1, 2]

// Modify the result cache.
values.cache.set(object, ['a', 'b']);
values(object);
// => ['a', 'b']

// Replace `_.memoize.Cache`.
_.memoize.Cache = WeakMap;
```

_.negate(predicate)

Creates a function that negates the result of the predicate `func`. The `func` predicate is invoked with the `this` binding and arguments of the created function.

Since

3.0.0

Arguments

`predicate` ***(Function)***: The predicate to negate.

Returns

(Function): Returns the new negated function.

Example

```
function isEven(n) {
  return n % 2 == 0;
}

_.filter([1, 2, 3, 4, 5, 6], _.negate(isEven));
// => [1, 3, 5]
```

_.once(func)

Creates a function that is restricted to invoking `func` once. Repeat calls to the function return the value of the first invocation. The `func` is invoked with the `this` binding and arguments of the created function.

Since

0.1.0

Arguments

`func` ***(Function)***: The function to restrict.

Returns

(Function): Returns the new restricted function.

Example

```
var initialize = _.once(createApplication);
initialize();
initialize();
// => `createApplication` is invoked once
```

.overArgs(func, [transforms=[.identity]])

Creates a function that invokes `func` with its arguments transformed.

Since

4.0.0

Arguments

`func` ***(Function)***: The function to wrap.
`[transforms=[_.identity]]` ***(...(Function|Function[]))***: The argument transforms.

Returns

(Function): Returns the new function.

Example

```
function doubled(n) {
  return n * 2;
}

function square(n) {
  return n * n;
}

var func = _.overArgs(function(x, y) {
  return [x, y];
}, [square, doubled]);

func(9, 3);
// => [81, 6]

func(10, 5);
// => [100, 10]
```

`_.partial(func, [partials])`

Creates a function that invokes `func` with `partials` prepended to the arguments it receives. This method is like `_.bind` except it does **not** alter the `this` binding.

The `_.partial.placeholder` value, which defaults to `_` in monolithic builds, may be used as a placeholder for partially applied arguments.

Note: This method doesn't set the "length" property of partially applied functions.

Since

0.2.0

Arguments

`func` ***(Function)***: The function to partially apply arguments to.
`[partials]` ***(...*)***: The arguments to be partially applied.

Returns

(Function): Returns the new partially applied function.

Example

```
function greet(greeting, name) {
  return greeting + ' ' + name;
}

var sayHelloTo = _.partial(greet, 'hello');
sayHelloTo('fred');
// => 'hello fred'

// Partially applied with placeholders.
var greetFred = _.partial(greet, _, 'fred');
greetFred('hi');
// => 'hi fred'
```

`_.partialRight(func, [partials])`

This method is like `_.partial` except that partially applied arguments are appended to the arguments it receives.

The `_.partialRight.placeholder` value, which defaults to `_` in monolithic builds, may be used as a placeholder for partially applied arguments.

Note: This method doesn't set the "length" property of partially applied functions.

Since

1.0.0

Arguments

`func` ***(Function)***: The function to partially apply arguments to.
`[partials]` ***(...*)***: The arguments to be partially applied.

Returns

(Function): Returns the new partially applied function.

Example

```
function greet(greeting, name) {
  return greeting + ' ' + name;
}

var greetFred = _.partialRight(greet, 'fred');
greetFred('hi');
// => 'hi fred'

// Partially applied with placeholders.
var sayHelloTo = _.partialRight(greet, 'hello', _);
sayHelloTo('fred');
// => 'hello fred'
```

_.rearg(func, indexes)

Creates a function that invokes `func` with arguments arranged according to the specified `indexes` where the argument value at the first index is provided as the first argument, the argument value at the second index is provided as the second argument, and so on.

Since

3.0.0

Arguments

`func` ***(Function)***: The function to rearrange arguments for.
`indexes` ***(...(number|number[]))***: The arranged argument indexes.

Returns

(Function): Returns the new function.

Example

```
var rearged = _.rearg(function(a, b, c) {
  return [a, b, c];
}, [2, 0, 1]);

rearged('b', 'c', 'a')
// => ['a', 'b', 'c']
```

_.rest(func, [start=func.length-1])

Creates a function that invokes `func` with the `this` binding of the created function and arguments from `start` and beyond provided as an array.

Note: This method is based on the rest parameter (https://mdn.io/rest_parameters).

Since

4.0.0

Arguments

`func` ***`(Function)`***: The function to apply a rest parameter to.
`[start=func.length-1]` ***`(number)`***: The start position of the rest parameter.

Returns

`(Function)`: Returns the new function.

Example

```
var say = _.rest(function(what, names) {
  return what + ' ' + _.initial(names).join(', ') +
    (_.size(names) > 1 ? ', & ' : '') + _.last(names);
});

say('hello', 'fred', 'barney', 'pebbles');
// => 'hello fred, barney, & pebbles'
```

`_.spread(func, [start=0])`

Creates a function that invokes `func` with the `this` binding of the create function and an array of arguments much like `Function#apply` (http://www.ecma-international.org/ecma-262/7.0/#sec-function.prototype.apply).

Note: This method is based on the spread operator (https://mdn.io/spread_operator).

Since

3.2.0

Arguments

`func` ***`(Function)`***: The function to spread arguments over.
`[start=0]` ***`(number)`***: The start position of the spread.

Returns

`(Function)`: Returns the new function.

Example

```
var say = _.spread(function(who, what) {
  return who + ' says ' + what;
});

say(['fred', 'hello']);
// => 'fred says hello'

var numbers = Promise.all([
  Promise.resolve(40),
  Promise.resolve(36)
]);

numbers.then(_.spread(function(x, y) {
  return x + y;
}));
// => a Promise of 76
```

`_.throttle(func, [wait=0], [options={}])`

Creates a throttled function that only invokes `func` at most once per every `wait` milliseconds. The throttled function comes with a `cancel` method to cancel delayed `func` invocations and a `flush` method to immediately invoke them. Provide `options` to indicate whether `func` should be invoked on the leading and/or trailing edge of the `wait` timeout. The `func` is invoked with the last arguments provided to the throttled function. Subsequent calls to the throttled function return the result of the last `func` invocation.

Note: If `leading` and `trailing` options are `true`, `func` is invoked on the trailing edge of the timeout only if the throttled function is invoked more than once during the `wait` timeout.

If `wait` is `0` and `leading` is `false`, `func` invocation is deferred until to the next tick, similar to `setTimeout` with a timeout of `0`.

See David Corbacho's article (https://css-tricks.com/debouncing-throttling-explained-examples/) for details over the differences between `_.throttle` and `_.debounce`.

Since

0.1.0

Arguments

`func` ***(Function)***: The function to throttle.
`[wait=0]` ***(number)***: The number of milliseconds to throttle invocations to.
`[options={}]` ***(Object)***: The options object.

[options.leading=true] ***(boolean)***: Specify invoking on the leading edge of the timeout.
[options.trailing=true] ***(boolean)***: Specify invoking on the trailing edge of the timeout.

Returns

(Function): Returns the new throttled function.

Example

```
// Avoid excessively updating the position while scrolling.
jQuery(window).on('scroll', _.throttle(updatePosition, 100));

// Invoke `renewToken` when the click event is fired, but not more than once every 5 m
var throttled = _.throttle(renewToken, 300000, { 'trailing': false });
jQuery(element).on('click', throttled);

// Cancel the trailing throttled invocation.
jQuery(window).on('popstate', throttled.cancel);
```

`_.unary(func)`

Creates a function that accepts up to one argument, ignoring any additional arguments.

Since

4.0.0

Arguments

func ***(Function)***: The function to cap arguments for.

Returns

(Function): Returns the new capped function.

Example

```
_.map(['6', '8', '10'], _.unary(parseInt));
// => [6, 8, 10]
```

`_.wrap(value, [wrapper=identity])`

Creates a function that provides `value` to `wrapper` as its first argument. Any additional arguments provided to the function are appended to those provided to the `wrapper`. The wrapper is invoked with the `this` binding of the created function.

Since

0.1.0

Arguments

`value` ***`(*)`***: The value to wrap.
`[wrapper=identity]` ***`(Function)`***: The wrapper function.

Returns

`(Function)`: Returns the new function.

Example

```
var p = _.wrap(_.escape, function(func, text) {
  return '<p>' + func(text) + '</p>';
});

p('fred, barney, & pebbles');
// => '<p>fred, barney, &amp; pebbles</p>'
```

“Lang” Methods

`_.castArray(value)`

Casts `value` as an array if it's not one.

Since

4.4.0

Arguments

`value` ***`(*)`***: The value to inspect.

Returns

`(Array)`: Returns the cast array.

Example

```
_.castArray(1);
// => [1]

_.castArray({ 'a': 1 });
// => [{ 'a': 1 }]

_.castArray('abc');
// => ['abc']

_.castArray(null);
// => [null]

_.castArray(undefined);
// => [undefined]

_.castArray();
// => []

var array = [1, 2, 3];
console.log(_.castArray(array) === array);
// => true
```

_.clone(value)

Creates a shallow clone of `value`.

Note: This method is loosely based on the structured clone algorithm (https://mdn.io/Structured_clone_algorithm) and supports cloning arrays, array buffers, booleans, date objects, maps, numbers, `Object` objects, regexes, sets, strings, symbols, and typed arrays. The own enumerable properties of `arguments` objects are cloned as plain objects. An empty object is returned for uncloneable values such as error objects, functions, DOM nodes, and WeakMaps.

Since

0.1.0

Arguments

`value` ***(*)***: The value to clone.

Returns

(*): Returns the cloned value.

Example

```
var objects = [{ 'a': 1 }, { 'b': 2 }];

var shallow = _.clone(objects);
console.log(shallow[0] === objects[0]);
// => true
```

_.cloneDeep(value)

This method is like `_.clone` except that it recursively clones `value`.

Since

1.0.0

Arguments

value ***(*)***: The value to recursively clone.

Returns

(*): Returns the deep cloned value.

Example

```
var objects = [{ 'a': 1 }, { 'b': 2 }];

var deep = _.cloneDeep(objects);
console.log(deep[0] === objects[0]);
// => false
```

_.cloneDeepWith(value, [customizer])

This method is like `_.cloneWith` except that it recursively clones `value`.

Since

4.0.0

Arguments

value ***(*)***: The value to recursively clone.
[customizer] ***(Function)***: The function to customize cloning.

Returns

(*): Returns the deep cloned value.

Example

```
function customizer(value) {
  if (_.isElement(value)) {
    return value.cloneNode(true);
  }
}

var el = _.cloneDeepWith(document.body, customizer);

console.log(el === document.body);
// => false
console.log(el.nodeName);
// => 'BODY'
console.log(el.childNodes.length);
// => 20
```

`_.cloneWith(value, [customizer])`

This method is like `_.clone` except that it accepts `customizer` which is invoked to produce the cloned value. If `customizer` returns `undefined`, cloning is handled by the method instead. The `customizer` is invoked with up to four arguments; *(value [, index|key, object, stack])*.

Since

4.0.0

Arguments

`value` ***(*)***: The value to clone.
`[customizer]` ***(Function)***: The function to customize cloning.

Returns

(*): Returns the cloned value.

Example

```
function customizer(value) {
  if (_.isElement(value)) {
    return value.cloneNode(false);
  }
}

var el = _.cloneWith(document.body, customizer);

console.log(el === document.body);
// => false
console.log(el.nodeName);
// => 'BODY'
console.log(el.childNodes.length);
// => 0
```

_.conformsTo(object, source)

Checks if `object` conforms to `source` by invoking the predicate properties of `source` with the corresponding property values of `object`.

Note: This method is equivalent to `_.conforms` when `source` is partially applied.

Since

4.14.0

Arguments

`object` ***(Object)***: The object to inspect.
`source` ***(Object)***: The object of property predicates to conform to.

Returns

(boolean): Returns `true` if `object` conforms, else `false`.

Example

```
var object = { 'a': 1, 'b': 2 };

_.conformsTo(object, { 'b': function(n) { return n > 1; } });
// => true

_.conformsTo(object, { 'b': function(n) { return n > 2; } });
// => false
```

_.eq(value, other)

Performs a `SameValueZero` (http://ecma-international.org/ecma-262/7.0/#sec-samevaluezero) comparison between two values to determine if they are equivalent.

Since

4.0.0

Arguments

`value` ***(*)***: The value to compare.
`other` ***(*)***: The other value to compare.

Returns

(boolean): Returns `true` if the values are equivalent, else `false`.

Example

```
var object = { 'a': 1 };
var other = { 'a': 1 };

_.eq(object, object);
// => true

_.eq(object, other);
// => false

_.eq('a', 'a');
// => true

_.eq('a', Object('a'));
// => false

_.eq(NaN, NaN);
// => true
```

_.gt(value, other)

Checks if `value` is greater than `other`.

Since

3.9.0

Arguments

`value` ***(*)***: The value to compare.
`other` ***(*)***: The other value to compare.

Returns

(boolean): Returns `true` if `value` is greater than `other`, else `false`.

Example

```
_.gt(3, 1);
// => true

_.gt(3, 3);
// => false

_.gt(1, 3);
// => false
```

`_.gte(value, other)`

Checks if `value` is greater than or equal to `other`.

Since

3.9.0

Arguments

`value` ***(*)***: The value to compare.
`other` ***(*)***: The other value to compare.

Returns

(boolean): Returns `true` if `value` is greater than or equal to `other`, else `false`.

Example

```
_.gte(3, 1);
// => true

_.gte(3, 3);
// => true

_.gte(1, 3);
// => false
```

_.isArguments(value)

Checks if `value` is likely an `arguments` object.

Since

0.1.0

Arguments

`value` *(*)*: The value to check.

Returns

(boolean): Returns `true` if `value` is an `arguments` object, else `false`.

Example

```
_.isArguments(function() { return arguments; }());
// => true

_.isArguments([1, 2, 3]);
// => false
```

_.isArray(value)

Checks if `value` is classified as an `Array` object.

Since

0.1.0

Arguments

`value` *(*)*: The value to check.

Returns

(boolean): Returns `true` if `value` is an array, else `false`.

Example

```
_.isArray([1, 2, 3]);
// => true

_.isArray(document.body.children);
// => false

_.isArray('abc');
// => false

_.isArray(_.noop);
// => false
```

_.isArrayBuffer(value)

Checks if `value` is classified as an `ArrayBuffer` object.

Since

4.3.0

Arguments

`value` ***(*)***: The value to check.

Returns

(boolean): Returns `true` if `value` is an array buffer, else `false`.

Example

```
_.isArrayBuffer(new ArrayBuffer(2));
// => true

_.isArrayBuffer(new Array(2));
// => false
```

_.isArrayLike(value)

Checks if `value` is array-like. A value is considered array-like if it's not a function and has a `value.length` that's an integer greater than or equal to `0` and less than or equal to `Number.MAX_SAFE_INTEGER`.

Since

4.0.0

Arguments

value *(*)*: The value to check.

Returns

(boolean): Returns `true` if `value` is array-like, else `false`.

Example

```
_.isArrayLike([1, 2, 3]);
// => true

_.isArrayLike(document.body.children);
// => true

_.isArrayLike('abc');
// => true

_.isArrayLike(_.noop);
// => false
```

`_.isArrayLikeObject(value)`

This method is like `_.isArrayLike` except that it also checks if `value` is an object.

Since

4.0.0

Arguments

value *(*)*: The value to check.

Returns

(boolean): Returns `true` if `value` is an array-like object, else `false`.

Example

```
_.isArrayLikeObject([1, 2, 3]);
// => true

_.isArrayLikeObject(document.body.children);
// => true

_.isArrayLikeObject('abc');
// => false

_.isArrayLikeObject(_.noop);
// => false
```

_.isBoolean(value)

Checks if `value` is classified as a boolean primitive or object.

Since

0.1.0

Arguments

`value` ***(*)***: The value to check.

Returns

(boolean): Returns `true` if `value` is a boolean, else `false`.

Example

```
_.isBoolean(false);
// => true

_.isBoolean(null);
// => false
```

_.isBuffer(value)

Checks if `value` is a buffer.

Since

4.3.0

Arguments

`value (*)`: The value to check.

Returns

(boolean): Returns `true` if `value` is a buffer, else `false`.

Example

```
_.isBuffer(new Buffer(2));
// => true

_.isBuffer(new Uint8Array(2));
// => false
```

_.isDate(value)

Checks if `value` is classified as a `Date` object.

Since

0.1.0

Arguments

`value (*)`: The value to check.

Returns

(boolean): Returns `true` if `value` is a date object, else `false`.

Example

```
_.isDate(new Date);
// => true

_.isDate('Mon April 23 2012');
// => false
```

_.isElement(value)

Checks if `value` is likely a DOM element.

Since

0.1.0

Arguments

`value` ***(*)***: The value to check.

Returns

(boolean): Returns `true` if `value` is a DOM element, else `false`.

Example

```
_.isElement(document.body);
// => true

_.isElement('<body>');
// => false
```

`_.isEmpty(value)`

Checks if `value` is an empty object, collection, map, or set.

Objects are considered empty if they have no own enumerable string keyed properties.

Array-like values such as `arguments` objects, arrays, buffers, strings, or jQuery-like collections are considered empty if they have a `length` of `0`. Similarly, maps and sets are considered empty if they have a `size` of `0`.

Since

0.1.0

Arguments

`value` ***(*)***: The value to check.

Returns

(boolean): Returns `true` if `value` is empty, else `false`.

Example

```
_.isEmpty(null);
// => true

_.isEmpty(true);
// => true

_.isEmpty(1);
// => true

_.isEmpty([1, 2, 3]);
// => false

_.isEmpty({ 'a': 1 });
// => false
```

_.isEqual(value, other)

Performs a deep comparison between two values to determine if they are equivalent.

Note: This method supports comparing arrays, array buffers, booleans, date objects, error objects, maps, numbers, `Object` objects, regexes, sets, strings, symbols, and typed arrays. `Object` objects are compared by their own, not inherited, enumerable properties. Functions and DOM nodes are compared by strict equality, i.e. ===.

Since

0.1.0

Arguments

`value` ***(*)***: The value to compare.
`other` ***(*)***: The other value to compare.

Returns

(boolean): Returns `true` if the values are equivalent, else `false`.

Example

```
var object = { 'a': 1 };
var other = { 'a': 1 };

_.isEqual(object, other);
// => true

object === other;
// => false
```

_.isEqualWith(value, other, [customizer])

This method is like `_.isEqual` except that it accepts `customizer` which is invoked to compare values. If `customizer` returns `undefined`, comparisons are handled by the method instead. The `customizer` is invoked with up to six arguments: *(objValue, othValue [, index|key, object, other, stack])*.

Since

4.0.0

Arguments

`value` ***(*)***: The value to compare.
`other` ***(*)***: The other value to compare.
`[customizer]` ***(Function)***: The function to customize comparisons.

Returns

(boolean): Returns `true` if the values are equivalent, else `false`.

Example

```
function isGreeting(value) {
  return /^h(?:i|ello)$/.test(value);
}

function customizer(objValue, othValue) {
  if (isGreeting(objValue) && isGreeting(othValue)) {
    return true;
  }
}

var array = ['hello', 'goodbye'];
var other = ['hi', 'goodbye'];

_.isEqualWith(array, other, customizer);
// => true
```

_.isError(value)

Checks if `value` is an `Error`, `EvalError`, `RangeError`, `ReferenceError`, `SyntaxError`, `TypeError`, or `URIError` object.

Since

3.0.0

Arguments

`value` *(*)*: The value to check.

Returns

(boolean): Returns `true` if `value` is an error object, else `false`.

Example

```
_.isError(new Error);
// => true

_.isError(Error);
// => false
```

_.isFinite(value)

Checks if `value` is a finite primitive number.

Note: This method is based on `Number.isFinite` (https://mdn.io/Number/isFinite).

Since

0.1.0

Arguments

`value` *(*)*: The value to check.

Returns

(boolean): Returns `true` if `value` is a finite number, else `false`.

Example

```
_.isFinite(3);
// => true

_.isFinite(Number.MIN_VALUE);
// => true

_.isFinite(Infinity);
// => false

_.isFinite('3');
// => false
```

_.isFunction(value)

Checks if `value` is classified as a `Function` object.

Since

0.1.0

Arguments

`value` ***(*)***: The value to check.

Returns

(boolean): Returns `true` if `value` is a function, else `false`.

Example

```
_.isFunction(_);
// => true

_.isFunction(/abc/);
// => false
```

_.isInteger(value)

Checks if `value` is an integer.

Note: This method is based on `Number.isInteger` (https://mdn.io/Number/isInteger).

Since

4.0.0

Arguments

value *(*)*: The value to check.

Returns

(boolean): Returns `true` if `value` is an integer, else `false`.

Example

```
_.isInteger(3);
// => true

_.isInteger(Number.MIN_VALUE);
// => false

_.isInteger(Infinity);
// => false

_.isInteger('3');
// => false
```

_.isLength(value)

Checks if `value` is a valid array-like length.

Note: This method is loosely based on `ToLength` (http://ecma-international.org/ecma-262/7.0/#sec-tolength).

Since

4.0.0

Arguments

value *(*)*: The value to check.

Returns

(boolean): Returns `true` if `value` is a valid length, else `false`.

Example

```
_.isLength(3);
// => true

_.isLength(Number.MIN_VALUE);
// => false

_.isLength(Infinity);
// => false

_.isLength('3');
// => false
```

_.isMap(value)

Checks if `value` is classified as a `Map` object.

Since

4.3.0

Arguments

`value` ***(*)***: The value to check.

Returns

(boolean): Returns `true` if `value` is a map, else `false`.

Example

```
_.isMap(new Map);
// => true

_.isMap(new WeakMap);
// => false
```

_.isMatch(object, source)

Performs a partial deep comparison between `object` and `source` to determine if `object` contains equivalent property values.

Note: This method is equivalent to `_.matches` when `source` is partially applied.

Partial comparisons will match empty array and empty object `source` values against any array or object value, respectively. See `_.isEqual` for a list of supported value comparisons.

Since

3.0.0

Arguments

`object` ***(Object)***: The object to inspect.
`source` ***(Object)***: The object of property values to match.

Returns

(boolean): Returns `true` if `object` is a match, else `false`.

Example

```
var object = { 'a': 1, 'b': 2 };

_.isMatch(object, { 'b': 2 });
// => true

_.isMatch(object, { 'b': 1 });
// => false
```

`_.isMatchWith(object, source, [customizer])`

This method is like `_.isMatch` except that it accepts `customizer` which is invoked to compare values. If `customizer` returns `undefined`, comparisons are handled by the method instead. The `customizer` is invoked with five arguments: *(objValue, srcValue, index|key, object, source)*.

Since

4.0.0

Arguments

`object` ***(Object)***: The object to inspect.
`source` ***(Object)***: The object of property values to match.
`[customizer]` ***(Function)***: The function to customize comparisons.

Returns

(boolean): Returns `true` if `object` is a match, else `false`.

Example

```
function isGreeting(value) {
  return /^h(?:i|ello)$/.test(value);
}

function customizer(objValue, srcValue) {
  if (isGreeting(objValue) && isGreeting(srcValue)) {
    return true;
  }
}

var object = { 'greeting': 'hello' };
var source = { 'greeting': 'hi' };

_.isMatchWith(object, source, customizer);
// => true
```

_.isNaN(value)

Checks if `value` is `NaN`.

Note: This method is based on `Number.isNaN` (https://mdn.io/Number/isNaN) and is not the same as global `isNaN` (https://mdn.io/isNaN) which returns `true` for `undefined` and other non-number values.

Since

0.1.0

Arguments

`value` ***(*)***: The value to check.

Returns

(boolean): Returns `true` if `value` is `NaN`, else `false`.

Example

```
_.isNaN(NaN);
// => true

_.isNaN(new Number(NaN));
// => true

isNaN(undefined);
// => true

_.isNaN(undefined);
// => false
```

_.isNative(value)

Checks if `value` is a pristine native function.

Note: This method can't reliably detect native functions in the presence of the core-js package because core-js circumvents this kind of detection. Despite multiple requests, the core-js maintainer has made it clear: any attempt to fix the detection will be obstructed. As a result, we're left with little choice but to throw an error. Unfortunately, this also affects packages, like babel-polyfill (https://www.npmjs.com/package/babel-polyfill), which rely on core-js.

Since

3.0.0

Arguments

`value` ***(*)***: The value to check.

Returns

(boolean): Returns `true` if `value` is a native function, else `false`.

Example

```
_.isNative(Array.prototype.push);
// => true

_.isNative(_);
// => false
```

_.isNil(value)

Checks if `value` is `null` or `undefined`.

Since

4.0.0

Arguments

`value` *(*)*: The value to check.

Returns

(boolean): Returns `true` if `value` is nullish, else `false`.

Example

```
_.isNil(null);
// => true

_.isNil(void 0);
// => true

_.isNil(NaN);
// => false
```

_.isNull(value)

Checks if `value` is `null`.

Since

0.1.0

Arguments

`value` *(*)*: The value to check.

Returns

(boolean): Returns `true` if `value` is `null`, else `false`.

Example

```
_.isNull(null);
// => true

_.isNull(void 0);
// => false
```

`_.isNumber(value)`

Checks if `value` is classified as a `Number` primitive or object.

Note: To exclude `Infinity`, `-Infinity`, and `NaN`, which are classified as numbers, use the `_.isFinite` method.

Since

0.1.0

Arguments

`value` *(*)*: The value to check.

Returns

(boolean): Returns `true` if `value` is a number, else `false`.

Example

```
_.isNumber(3);
// => true

_.isNumber(Number.MIN_VALUE);
// => true

_.isNumber(Infinity);
// => true

_.isNumber('3');
// => false
```

`_.isObject(value)`

Checks if `value` is the language type (http://www.ecma-international.org/ecma-262/7.0/#sec-ecmascript-language-types) of `Object`. *(e.g. arrays, functions, objects, regexes, `new Number(0)`, and `new String('')`)*

Since

0.1.0

Arguments

`value` *(*)*: The value to check.

Returns

(boolean): Returns `true` if `value` is an object, else `false`.

Example

```
_.isObject({});
// => true

_.isObject([1, 2, 3]);
// => true

_.isObject(_.noop);
// => true

_.isObject(null);
// => false
```

`_.isObjectLike(value)`

Checks if `value` is object-like. A value is object-like if it's not `null` and has a `typeof` result of "object".

Since

4.0.0

Arguments

`value` ***(*)***: The value to check.

Returns

(boolean): Returns `true` if `value` is object-like, else `false`.

Example

```
_.isObjectLike({});
// => true

_.isObjectLike([1, 2, 3]);
// => true

_.isObjectLike(_.noop);
// => false

_.isObjectLike(null);
// => false
```

_.isPlainObject(value)

Checks if `value` is a plain object, that is, an object created by the `Object` constructor or one with a `[[Prototype]]` of `null`.

Since

0.8.0

Arguments

`value` *(*)*: The value to check.

Returns

(boolean): Returns `true` if `value` is a plain object, else `false`.

Example

```
function Foo() {
  this.a = 1;
}

_.isPlainObject(new Foo);
// => false

_.isPlainObject([1, 2, 3]);
// => false

_.isPlainObject({ 'x': 0, 'y': 0 });
// => true

_.isPlainObject(Object.create(null));
// => true
```

_.isRegExp(value)

Checks if `value` is classified as a `RegExp` object.

Since

0.1.0

Arguments

`value (*)`: The value to check.

Returns

(boolean): Returns `true` if `value` is a regexp, else `false`.

Example

```
_.isRegExp(/abc/);
// => true

_.isRegExp('/abc/');
// => false
```

`_.isSafeInteger(value)`

Checks if `value` is a safe integer. An integer is safe if it's an IEEE-754 double precision number which isn't the result of a rounded unsafe integer.

Note: This method is based on `Number.isSafeInteger` (https://mdn.io/Number/isSafeInteger).

Since

4.0.0

Arguments

`value (*)`: The value to check.

Returns

(boolean): Returns `true` if `value` is a safe integer, else `false`.

Example

```
_.isSafeInteger(3);
// => true

_.isSafeInteger(Number.MIN_VALUE);
// => false

_.isSafeInteger(Infinity);
// => false

_.isSafeInteger('3');
// => false
```

_.isSet(value)

Checks if `value` is classified as a `Set` object.

Since

4.3.0

Arguments

`value` *(*)*: The value to check.

Returns

(boolean): Returns `true` if `value` is a set, else `false`.

Example

```
_.isSet(new Set);
// => true

_.isSet(new WeakSet);
// => false
```

_.isString(value)

Checks if `value` is classified as a `String` primitive or object.

Since

0.1.0

Arguments

`value` *(*)*: The value to check.

Returns

(boolean): Returns `true` if `value` is a string, else `false`.

Example

```
_.isString('abc');
// => true

_.isString(1);
// => false
```

_.isSymbol(value)

Checks if `value` is classified as a `Symbol` primitive or object.

Since

4.0.0

Arguments

`value` *(*)*: The value to check.

Returns

(boolean): Returns `true` if `value` is a symbol, else `false`.

Example

```
_.isSymbol(Symbol.iterator);
// => true

_.isSymbol('abc');
// => false
```

_.isTypedArray(value)

Checks if `value` is classified as a typed array.

Since

3.0.0

Arguments

`value` *(*)*: The value to check.

Returns

(boolean): Returns `true` if `value` is a typed array, else `false`.

Example

```
_.isTypedArray(new Uint8Array);
// => true

_.isTypedArray([]);
// => false
```

_.isUndefined(value)

Checks if `value` is `undefined`.

Since

0.1.0

Arguments

`value` ***(*)***: The value to check.

Returns

(boolean): Returns `true` if `value` is `undefined`, else `false`.

Example

```
_.isUndefined(void 0);
// => true

_.isUndefined(null);
// => false
```

_.isWeakMap(value)

Checks if `value` is classified as a `WeakMap` object.

Since

4.3.0

Arguments

`value` ***(*)***: The value to check.

Returns

(boolean): Returns `true` if `value` is a weak map, else `false`.

Example

```
_.isWeakMap(new WeakMap);
// => true

_.isWeakMap(new Map);
// => false
```

_.isWeakSet(value)

Checks if `value` is classified as a `WeakSet` object.

Since

4.3.0

Arguments

`value` ***(*)***: The value to check.

Returns

(boolean): Returns `true` if `value` is a weak set, else `false`.

Example

```
_.isWeakSet(new WeakSet);
// => true

_.isWeakSet(new Set);
// => false
```

_.lt(value, other)

Checks if `value` is less than `other`.

Since

3.9.0

Arguments

`value` ***(*)***: The value to compare.
`other` ***(*)***: The other value to compare.

Returns

(boolean): Returns `true` if `value` is less than `other`, else `false`.

Example

```
_.lt(1, 3);
// => true

_.lt(3, 3);
// => false

_.lt(3, 1);
// => false
```

_.lte(value, other)

Checks if `value` is less than or equal to `other`.

Since

3.9.0

Arguments

`value` ***(*)***: The value to compare.
`other` ***(*)***: The other value to compare.

Returns

(boolean): Returns `true` if `value` is less than or equal to `other`, else `false`.

Example

```
_.lte(1, 3);
// => true

_.lte(3, 3);
// => true

_.lte(3, 1);
// => false
```

_.toArray(value)

Converts `value` to an array.

Since

0.1.0

Arguments

`value` ***(*)***: The value to convert.

Returns

(Array): Returns the converted array.

Example

```
_.toArray({ 'a': 1, 'b': 2 });
// => [1, 2]

_.toArray('abc');
// => ['a', 'b', 'c']

_.toArray(1);
// => []

_.toArray(null);
// => []
```

`_.toFinite(value)`

Converts `value` to a finite number.

Since

4.12.0

Arguments

`value` ***(*)***: The value to convert.

Returns

(number): Returns the converted number.

Example

```
_.toFinite(3.2);
// => 3.2

_.toFinite(Number.MIN_VALUE);
// => 5e-324

_.toFinite(Infinity);
// => 1.7976931348623157e+308

_.toFinite('3.2');
// => 3.2
```

_.toInteger(value)

Converts `value` to an integer.

Note: This method is loosely based on `ToInteger` (http://www.ecma-international.org/ecma-262/7.0/#sec-tointeger).

Since

4.0.0

Arguments

value *(*)*: The value to convert.

Returns

(number): Returns the converted integer.

Example

```
_.toInteger(3.2);
// => 3

_.toInteger(Number.MIN_VALUE);
// => 0

_.toInteger(Infinity);
// => 1.7976931348623157e+308

_.toInteger('3.2');
// => 3
```

_.toLength(value)

Converts `value` to an integer suitable for use as the length of an array-like object.

Note: This method is based on `ToLength` (http://ecma-international.org/ecma-262/7.0/#sec-tolength).

Since

4.0.0

Arguments

`value` *(*)*: The value to convert.

Returns

(number): Returns the converted integer.

Example

```
_.toLength(3.2);
// => 3

_.toLength(Number.MIN_VALUE);
// => 0

_.toLength(Infinity);
// => 4294967295

_.toLength('3.2');
// => 3
```

_.toNumber(value)

Converts `value` to a number.

Since

4.0.0

Arguments

`value` *(*)*: The value to process.

Returns

(number): Returns the number.

Example

```
_.toNumber(3.2);
// => 3.2

_.toNumber(Number.MIN_VALUE);
// => 5e-324

_.toNumber(Infinity);
// => Infinity

_.toNumber('3.2');
// => 3.2
```

_.toPlainObject(value)

Converts `value` to a plain object flattening inherited enumerable string keyed properties of `value` to own properties of the plain object.

Since

3.0.0

Arguments

value ***(*)***: The value to convert.

Returns

(Object): Returns the converted plain object.

Example

```
function Foo() {
  this.b = 2;
}

Foo.prototype.c = 3;

_.assign({ 'a': 1 }, new Foo);
// => { 'a': 1, 'b': 2 }

_.assign({ 'a': 1 }, _.toPlainObject(new Foo));
// => { 'a': 1, 'b': 2, 'c': 3 }
```

_.toSafeInteger(value)

Converts `value` to a safe integer. A safe integer can be compared and represented correctly.

Since

4.0.0

Arguments

`value` ***(*)***: The value to convert.

Returns

(number): Returns the converted integer.

Example

```
_.toSafeInteger(3.2);
// => 3

_.toSafeInteger(Number.MIN_VALUE);
// => 0

_.toSafeInteger(Infinity);
// => 9007199254740991

_.toSafeInteger('3.2');
// => 3
```

_.toString(value)

Converts `value` to a string. An empty string is returned for `null` and `undefined` values. The sign of `-0` is preserved.

Since

4.0.0

Arguments

`value` ***(*)***: The value to convert.

Returns

(string): Returns the converted string.

Example

```
_.toString(null);
// => ''

_.toString(-0);
// => '-0'

_.toString([1, 2, 3]);
// => '1,2,3'
```

“Math” Methods

_.add(augend, addend)

Adds two numbers.

Since

3.4.0

Arguments

augend ***(number)***: The first number in an addition.
addend ***(number)***: The second number in an addition.

Returns

(number): Returns the total.

Example

```
_.add(6, 4);
// => 10
```

_.ceil(number, [precision=0])

Computes `number` rounded up to `precision`.

Since

3.10.0

Arguments

number ***(number)***: The number to round up.
[precision=0] ***(number)***: The precision to round up to.

Returns

(number): Returns the rounded up number.

Example

```
_.ceil(4.006);
// => 5

_.ceil(6.004, 2);
// => 6.01

_.ceil(6040, -2);
// => 6100
```

_.divide(dividend, divisor)

Divide two numbers.

Since

4.7.0

Arguments

dividend ***(number)***: The first number in a division.
divisor ***(number)***: The second number in a division.

Returns

(number): Returns the quotient.

Example

```
_.divide(6, 4);
// => 1.5
```

_.floor(number, [precision=0])

Computes `number` rounded down to `precision`.

Since

3.10.0

Arguments

number ***(number)***: The number to round down.
[precision=0] ***(number)***: The precision to round down to.

Returns

(number): Returns the rounded down number.

Example

```
_.floor(4.006);
// => 4

_.floor(0.046, 2);
// => 0.04

_.floor(4060, -2);
// => 4000
```

_.max(array)

Computes the maximum value of `array`. If `array` is empty or falsey, `undefined` is returned.

Since

0.1.0

Arguments

array ***(Array)***: The array to iterate over.

Returns

(*): Returns the maximum value.

Example

```
_.max([4, 2, 8, 6]);
// => 8

_.max([]);
// => undefined
```

.maxBy(array, [iteratee=.identity])

This method is like `_.max` except that it accepts `iteratee` which is invoked for each element in `array` to generate the criterion by which the value is ranked. The iteratee is invoked with one argument: *(value)*.

Since

4.0.0

Arguments

array ***(Array)***: The array to iterate over.
[iteratee=_.identity] ***(Function)***: The iteratee invoked per element.

Returns

(*): Returns the maximum value.

Example

```
var objects = [{ 'n': 1 }, { 'n': 2 }];

_.maxBy(objects, function(o) { return o.n; });
// => { 'n': 2 }

// The `_.property` iteratee shorthand.
_.maxBy(objects, 'n');
// => { 'n': 2 }
```

_.mean(array)

Computes the mean of the values in `array`.

Since

4.0.0

Arguments

array ***(Array)***: The array to iterate over.

Returns

(number): Returns the mean.

Example

```
_.mean([4, 2, 8, 6]);
// => 5
```

.meanBy(array, [iteratee=.identity])

This method is like `_.mean` except that it accepts `iteratee` which is invoked for each element in `array` to generate the value to be averaged. The iteratee is invoked with one argument: *(value)*.

Since

4.7.0

Arguments

array ***(Array)***: The array to iterate over.
[iteratee=_.identity] ***(Function)***: The iteratee invoked per element.

Returns

(number): Returns the mean.

Example

```
var objects = [{ 'n': 4 }, { 'n': 2 }, { 'n': 8 }, { 'n': 6 }];

_.meanBy(objects, function(o) { return o.n; });
// => 5

// The `_.property` iteratee shorthand.
_.meanBy(objects, 'n');
// => 5
```

_.min(array)

Computes the minimum value of `array`. If `array` is empty or falsey, `undefined` is returned.

Since

0.1.0

Arguments

array ***(Array)***: The array to iterate over.

Returns

(*): Returns the minimum value.

Example

```
_.min([4, 2, 8, 6]);
// => 2

_.min([]);
// => undefined
```

.minBy(array, [iteratee=.identity])

This method is like `_.min` except that it accepts `iteratee` which is invoked for each element in `array` to generate the criterion by which the value is ranked. The iteratee is invoked with one argument: *(value)*.

Since

4.0.0

Arguments

array ***(Array)***: The array to iterate over.
[iteratee=_.identity] ***(Function)***: The iteratee invoked per element.

Returns

(*): Returns the minimum value.

Example

```
var objects = [{ 'n': 1 }, { 'n': 2 }];

_.minBy(objects, function(o) { return o.n; });
// => { 'n': 1 }

// The `_.property` iteratee shorthand.
_.minBy(objects, 'n');
// => { 'n': 1 }
```

_.multiply(multiplier, multiplicand)

Multiply two numbers.

Since

4.7.0

Arguments

`multiplier` ***(number)***: The first number in a multiplication.
`multiplicand` ***(number)***: The second number in a multiplication.

Returns

(number): Returns the product.

Example

```
_.multiply(6, 4);
// => 24
```

`_.round(number, [precision=0])`

Computes `number` rounded to `precision`.

Since

3.10.0

Arguments

`number` ***(number)***: The number to round.
`[precision=0]` ***(number)***: The precision to round to.

Returns

(number): Returns the rounded number.

Example

```
_.round(4.006);
// => 4

_.round(4.006, 2);
// => 4.01

_.round(4060, -2);
// => 4100
```

`_.subtract(minuend, subtrahend)`

Subtract two numbers.

Since

4.0.0

Arguments

`minuend` *(number)*: The first number in a subtraction.
`subtrahend` *(number)*: The second number in a subtraction.

Returns

(number): Returns the difference.

Example

```
_.subtract(6, 4);
// => 2
```

`_.sum(array)`

Computes the sum of the values in `array`.

Since

3.4.0

Arguments

`array` *(Array)*: The array to iterate over.

Returns

(number): Returns the sum.

Example

```
_.sum([4, 2, 8, 6]);
// => 20
```

`_.sumBy(array, [iteratee=_.identity])`

This method is like `_.sum` except that it accepts `iteratee` which is invoked for each element in `array` to generate the value to be summed. The iteratee is invoked with one argument: *(value)*.

Since

4.0.0

Arguments

array ***(Array)***: The array to iterate over.
[iteratee=_.identity] ***(Function)***: The iteratee invoked per element.

Returns

(number): Returns the sum.

Example

```
var objects = [{ 'n': 4 }, { 'n': 2 }, { 'n': 8 }, { 'n': 6 }];

_.sumBy(objects, function(o) { return o.n; });
// => 20

// The `_.property` iteratee shorthand.
_.sumBy(objects, 'n');
// => 20
```

“Number” Methods

_.clamp(number, [lower], upper)

Clamps `number` within the inclusive `lower` and `upper` bounds.

Since

4.0.0

Arguments

number ***(number)***: The number to clamp.
[lower] ***(number)***: The lower bound.
upper ***(number)***: The upper bound.

Returns

(number): Returns the clamped number.

Example

```
_.clamp(-10, -5, 5);
// => -5

_.clamp(10, -5, 5);
// => 5
```

_.inRange(number, [start=0], end)

Checks if n is between `start` and up to, but not including, `end`. If `end` is not specified, it's set to `start` with `start` then set to `0`. If `start` is greater than `end` the params are swapped to support negative ranges.

Since

3.3.0

Arguments

`number` ***(number)***: The number to check.
`[start=0]` ***(number)***: The start of the range.
`end` ***(number)***: The end of the range.

Returns

(boolean): Returns `true` if `number` is in the range, else `false`.

Example

```
_.inRange(3, 2, 4);
// => true

_.inRange(4, 8);
// => true

_.inRange(4, 2);
// => false

_.inRange(2, 2);
// => false

_.inRange(1.2, 2);
// => true

_.inRange(5.2, 4);
// => false

_.inRange(-3, -2, -6);
// => true
```

_.random([lower=0], [upper=1], [floating])

Produces a random number between the inclusive `lower` and `upper` bounds. If only one argument is provided a number between `0` and the given number is returned. If `floating` is `true`, or either `lower` or `upper` are floats, a floating-point number is returned instead of an integer.

Note: JavaScript follows the IEEE-754 standard for resolving floating-point values which can produce unexpected results.

Since

0.7.0

Arguments

`[lower=0]` ***(number)***: The lower bound.
`[upper=1]` ***(number)***: The upper bound.
`[floating]` ***(boolean)***: Specify returning a floating-point number.

Returns

(number): Returns the random number.

Example

```
_.random(0, 5);
// => an integer between 0 and 5

_.random(5);
// => also an integer between 0 and 5

_.random(5, true);
// => a floating-point number between 0 and 5

_.random(1.2, 5.2);
// => a floating-point number between 1.2 and 5.2
```

“Object” Methods

_.assign(object, [sources])

Assigns own enumerable string keyed properties of source objects to the destination object. Source objects are applied from left to right. Subsequent sources overwrite property assignments of previous sources.

Note: This method mutates `object` and is loosely based on `Object.assign` (https://mdn.io/Object/assign).

Since

0.10.0

Arguments

`object` ***(Object)***: The destination object.
`[sources]` ***(...Object)***: The source objects.

Returns

(Object): Returns `object`.

Example

```
function Foo() {
  this.a = 1;
}

function Bar() {
  this.c = 3;
}

Foo.prototype.b = 2;
Bar.prototype.d = 4;

_.assign({ 'a': 0 }, new Foo, new Bar);
// => { 'a': 1, 'c': 3 }
```

_.assignIn(object, [sources])

This method is like `_.assign` except that it iterates over own and inherited source properties.

Note: This method mutates `object`.

Since

4.0.0

Aliases

_.extend

Arguments

`object` ***(Object)***: The destination object.
`[sources]` ***(...Object)***: The source objects.

Returns

(Object): Returns `object`.

Example

```
function Foo() {
  this.a = 1;
}

function Bar() {
  this.c = 3;
}

Foo.prototype.b = 2;
Bar.prototype.d = 4;

_.assignIn({ 'a': 0 }, new Foo, new Bar);
// => { 'a': 1, 'b': 2, 'c': 3, 'd': 4 }
```

_.assignInWith(object, sources, [customizer])

This method is like `_.assignIn` except that it accepts `customizer` which is invoked to produce the assigned values. If `customizer` returns `undefined`, assignment is handled by the method instead. The `customizer` is invoked with five arguments: *(objValue, srcValue, key, object, source)*.

Note: This method mutates `object`.

Since

4.0.0

Aliases

_.extendWith

Arguments

`object` ***(Object)***: The destination object.
`sources` ***(...Object)***: The source objects.
`[customizer]` ***(Function)***: The function to customize assigned values.

Returns

(Object): Returns `object`.

Example

```
function customizer(objValue, srcValue) {
  return _.isUndefined(objValue) ? srcValue : objValue;
}

var defaults = _.partialRight(_.assignInWith, customizer);

defaults({ 'a': 1 }, { 'b': 2 }, { 'a': 3 });
// => { 'a': 1, 'b': 2 }
```

_.assignWith(object, sources, [customizer])

This method is like `_.assign` except that it accepts `customizer` which is invoked to produce the assigned values. If `customizer` returns `undefined`, assignment is handled by the method instead. The `customizer` is invoked with five arguments: *(objValue, srcValue, key, object, source)*.

Note: This method mutates `object`.

Since

4.0.0

Arguments

`object` ***(Object)***: The destination object.
`sources` ***(...Object)***: The source objects.
`[customizer]` ***(Function)***: The function to customize assigned values.

Returns

(Object): Returns `object`.

Example

```
function customizer(objValue, srcValue) {
  return _.isUndefined(objValue) ? srcValue : objValue;
}

var defaults = _.partialRight(_.assignWith, customizer);

defaults({ 'a': 1 }, { 'b': 2 }, { 'a': 3 });
// => { 'a': 1, 'b': 2 }
```

_.at(object, [paths])

Creates an array of values corresponding to `paths` of `object`.

Since

1.0.0

Arguments

`object` ***(Object)***: The object to iterate over.
`[paths]` ***(...(string|string[]))***: The property paths to pick.

Returns

(Array): Returns the picked values.

Example

```
var object = { 'a': [{ 'b': { 'c': 3 } }, 4] };

_.at(object, ['a[0].b.c', 'a[1]']);
// => [3, 4]
```

`_.create(prototype, [properties])`

Creates an object that inherits from the `prototype` object. If a `properties` object is given, its own enumerable string keyed properties are assigned to the created object.

Since

2.3.0

Arguments

`prototype` ***(Object)***: The object to inherit from.
`[properties]` ***(Object)***: The properties to assign to the object.

Returns

(Object): Returns the new object.

Example

```
function Shape() {
  this.x = 0;
  this.y = 0;
}

function Circle() {
  Shape.call(this);
}

Circle.prototype = _.create(Shape.prototype, {
  'constructor': Circle
});

var circle = new Circle;
circle instanceof Circle;
// => true

circle instanceof Shape;
// => true
```

_.defaults(object, [sources])

Assigns own and inherited enumerable string keyed properties of source objects to the destination object for all destination properties that resolve to `undefined`. Source objects are applied from left to right. Once a property is set, additional values of the same property are ignored.

Note: This method mutates `object`.

Since

0.1.0

Arguments

`object` ***(Object)***: The destination object.
`[sources]` ***(...Object)***: The source objects.

Returns

(Object): Returns `object`.

Example

```
_.defaults({ 'a': 1 }, { 'b': 2 }, { 'a': 3 });
// => { 'a': 1, 'b': 2 }
```

`_.defaultsDeep(object, [sources])`

This method is like `_.defaults` except that it recursively assigns default properties.

Note: This method mutates `object`.

Since

3.10.0

Arguments

`object` ***(Object)***: The destination object.
`[sources]` ***(...Object)***: The source objects.

Returns

(Object): Returns `object`.

Example

```
_.defaultsDeep({ 'a': { 'b': 2 } }, { 'a': { 'b': 1, 'c': 3 } });
// => { 'a': { 'b': 2, 'c': 3 } }
```

`_.findKey(object, [predicate=_.identity])`

This method is like `_.find` except that it returns the key of the first element `predicate` returns truthy for instead of the element itself.

Since

1.1.0

Arguments

`object` ***(Object)***: The object to inspect.
`[predicate=_.identity]` ***(Function)***: The function invoked per iteration.

Returns

(*): Returns the key of the matched element, else `undefined`.

Example

```
var users = {
  'barney':  { 'age': 36, 'active': true },
  'fred':    { 'age': 40, 'active': false },
  'pebbles': { 'age': 1,  'active': true }
};

_.findKey(users, function(o) { return o.age < 40; });
// => 'barney' (iteration order is not guaranteed)

// The `_.matches` iteratee shorthand.
_.findKey(users, { 'age': 1, 'active': true });
// => 'pebbles'

// The `_.matchesProperty` iteratee shorthand.
_.findKey(users, ['active', false]);
// => 'fred'

// The `_.property` iteratee shorthand.
_.findKey(users, 'active');
// => 'barney'
```

.findLastKey(object, [predicate=.identity])

This method is like `_.findKey` except that it iterates over elements of a collection in the opposite order.

Since

2.0.0

Arguments

object ***(Object)***: The object to inspect.
[predicate=_.identity] ***(Function)***: The function invoked per iteration.

Returns

(*): Returns the key of the matched element, else `undefined`.

Example

```
var users = {
  'barney':  { 'age': 36, 'active': true },
  'fred':    { 'age': 40, 'active': false },
  'pebbles': { 'age': 1,  'active': true }
};

_.findLastKey(users, function(o) { return o.age < 40; });
// => returns 'pebbles' assuming `_.findKey` returns 'barney'

// The `_.matches` iteratee shorthand.
_.findLastKey(users, { 'age': 36, 'active': true });
// => 'barney'

// The `_.matchesProperty` iteratee shorthand.
_.findLastKey(users, ['active', false]);
// => 'fred'

// The `_.property` iteratee shorthand.
_.findLastKey(users, 'active');
// => 'pebbles'
```

.forIn(object, [iteratee=.identity])

Iterates over own and inherited enumerable string keyed properties of an object and invokes `iteratee` for each property. The iteratee is invoked with three arguments: *(value, key, object)*. Iteratee functions may exit iteration early by explicitly returning `false`.

Since

0.3.0

Arguments

`object` ***(Object)***: The object to iterate over.
`[iteratee=_.identity]` ***(Function)***: The function invoked per iteration.

Returns

(Object): Returns `object`.

Example

```
function Foo() {
  this.a = 1;
  this.b = 2;
}

Foo.prototype.c = 3;

_.forIn(new Foo, function(value, key) {
  console.log(key);
});
// => Logs 'a', 'b', then 'c' (iteration order is not guaranteed).
```

.forInRight(object, [iteratee=.identity])

This method is like `_.forIn` except that it iterates over properties of `object` in the opposite order.

Since

2.0.0

Arguments

`object` ***(Object)***: The object to iterate over.
`[iteratee=_.identity]` ***(Function)***: The function invoked per iteration.

Returns

(Object): Returns `object`.

Example

```
function Foo() {
  this.a = 1;
  this.b = 2;
}

Foo.prototype.c = 3;

_.forInRight(new Foo, function(value, key) {
  console.log(key);
});
// => Logs 'c', 'b', then 'a' assuming `_.forIn` logs 'a', 'b', then 'c'.
```

.forOwn(object, [iteratee=.identity])

Iterates over own enumerable string keyed properties of an object and invokes `iteratee` for each property. The iteratee is invoked with three arguments: *(value, key, object)*. Iteratee functions may exit iteration early by explicitly returning `false`.

Since

0.3.0

Arguments

`object` ***(Object)***: The object to iterate over.
`[iteratee=_.identity]` ***(Function)***: The function invoked per iteration.

Returns

(Object): Returns `object`.

Example

```
function Foo() {
  this.a = 1;
  this.b = 2;
}

Foo.prototype.c = 3;

_.forOwn(new Foo, function(value, key) {
  console.log(key);
});
// => Logs 'a' then 'b' (iteration order is not guaranteed).
```

`_.forOwnRight(object, [iteratee=_.identity])`

This method is like `_.forOwn` except that it iterates over properties of `object` in the opposite order.

Since

2.0.0

Arguments

`object` ***(Object)***: The object to iterate over.
`[iteratee=_.identity]` ***(Function)***: The function invoked per iteration.

Returns

(Object): Returns `object`.

Example

```
function Foo() {
  this.a = 1;
  this.b = 2;
}

Foo.prototype.c = 3;

_.forOwnRight(new Foo, function(value, key) {
  console.log(key);
});
// => Logs 'b' then 'a' assuming `_.forOwn` logs 'a' then 'b'.
```

_.functions(object)

Creates an array of function property names from own enumerable properties of `object`.

Since

0.1.0

Arguments

object ***(Object)***: The object to inspect.

Returns

(Array): Returns the function names.

Example

```
function Foo() {
  this.a = _.constant('a');
  this.b = _.constant('b');
}

Foo.prototype.c = _.constant('c');

_.functions(new Foo);
// => ['a', 'b']
```

_.functionsIn(object)

Creates an array of function property names from own and inherited enumerable properties of `object`.

Since

4.0.0

Arguments

`object` ***(Object)***: The object to inspect.

Returns

(Array): Returns the function names.

Example

```
function Foo() {
  this.a = _.constant('a');
  this.b = _.constant('b');
}

Foo.prototype.c = _.constant('c');

_.functionsIn(new Foo);
// => ['a', 'b', 'c']
```

`_.get(object, path, [defaultValue])`

Gets the value at `path` of `object`. If the resolved value is `undefined`, the `defaultValue` is returned in its place.

Since

3.7.0

Arguments

`object` ***(Object)***: The object to query.
`path` ***(Array|string)***: The path of the property to get.
`[defaultValue]` ***(*)***: The value returned for **`undefined`** resolved values.

Returns

(*): Returns the resolved value.

Example

```
var object = { 'a': [{ 'b': { 'c': 3 } }] };

_.get(object, 'a[0].b.c');
// => 3

_.get(object, ['a', '0', 'b', 'c']);
// => 3

_.get(object, 'a.b.c', 'default');
// => 'default'
```

_.has(object, path)

Checks if `path` is a direct property of `object`.

Since

0.1.0

Arguments

`object` ***(Object)***: The object to query.
`path` ***(Array|string)***: The path to check.

Returns

(boolean): Returns `true` if `path` exists, else `false`.

Example

```
var object = { 'a': { 'b': 2 } };
var other = _.create({ 'a': _.create({ 'b': 2 }) });

_.has(object, 'a');
// => true

_.has(object, 'a.b');
// => true

_.has(object, ['a', 'b']);
// => true

_.has(other, 'a');
// => false
```

_.hasIn(object, path)

Checks if `path` is a direct or inherited property of `object`.

Since

4.0.0

Arguments

`object` ***(Object)***: The object to query.
`path` ***(Array|string)***: The path to check.

Returns

(boolean): Returns `true` if `path` exists, else `false`.

Example

```
var object = _.create({ 'a': _.create({ 'b': 2 }) });

_.hasIn(object, 'a');
// => true

_.hasIn(object, 'a.b');
// => true

_.hasIn(object, ['a', 'b']);
// => true

_.hasIn(object, 'b');
// => false
```

_.invert(object)

Creates an object composed of the inverted keys and values of `object`. If `object` contains duplicate values, subsequent values overwrite property assignments of previous values.

Since

0.7.0

Arguments

`object` ***(Object)***: The object to invert.

Returns

(Object): Returns the new inverted object.

Example

```
var object = { 'a': 1, 'b': 2, 'c': 1 };

_.invert(object);
// => { '1': 'c', '2': 'b' }
```

.invertBy(object, [iteratee=.identity])

This method is like `_.invert` except that the inverted object is generated from the results of running each element of `object` thru `iteratee`. The corresponding inverted value of each inverted key is an array of keys responsible for generating the inverted value. The iteratee is invoked with one argument: *(value)*.

Since

4.1.0

Arguments

object ***(Object)***: The object to invert.
[iteratee=_.identity] ***(Function)***: The iteratee invoked per element.

Returns

(Object): Returns the new inverted object.

Example

```
var object = { 'a': 1, 'b': 2, 'c': 1 };

_.invertBy(object);
// => { '1': ['a', 'c'], '2': ['b'] }

_.invertBy(object, function(value) {
  return 'group' + value;
});
// => { 'group1': ['a', 'c'], 'group2': ['b'] }
```

_.invoke(object, path, [args])

Invokes the method at `path` of `object`.

Since

4.0.0

Arguments

`object` ***(Object)***: The object to query.
`path` ***(Array|string)***: The path of the method to invoke.
`[args]` ***(...*)***: The arguments to invoke the method with.

Returns

(*): Returns the result of the invoked method.

Example

```
var object = { 'a': [{ 'b': { 'c': [1, 2, 3, 4] } }] };

_.invoke(object, 'a[0].b.c.slice', 1, 3);
// => [2, 3]
```

`_.keys(object)`

Creates an array of the own enumerable property names of `object`.

Note: Non-object values are coerced to objects. See the ES spec (http://ecma-international.org/ecma-262/7.0/#sec-object.keys) for more details.

Since

0.1.0

Arguments

`object` ***(Object)***: The object to query.

Returns

(Array): Returns the array of property names.

Example

```
function Foo() {
  this.a = 1;
  this.b = 2;
}

Foo.prototype.c = 3;

_.keys(new Foo);
// => ['a', 'b'] (iteration order is not guaranteed)

_.keys('hi');
// => ['0', '1']
```

_.keysIn(object)

Creates an array of the own and inherited enumerable property names of `object`.

Note: Non-object values are coerced to objects.

Since

3.0.0

Arguments

object ***(Object)***: The object to query.

Returns

(Array): Returns the array of property names.

Example

```
function Foo() {
  this.a = 1;
  this.b = 2;
}

Foo.prototype.c = 3;

_.keysIn(new Foo);
// => ['a', 'b', 'c'] (iteration order is not guaranteed)
```

.mapKeys(object, [iteratee=.identity])

The opposite of `_.mapValues`; this method creates an object with the same values as `object` and keys generated by running each own enumerable string keyed property of `object` thru `iteratee`. The iteratee is invoked with three arguments: *(value, key, object)*.

Since

3.8.0

Arguments

`object` ***(Object)***: The object to iterate over.
`[iteratee=_.identity]` ***(Function)***: The function invoked per iteration.

Returns

(Object): Returns the new mapped object.

Example

```
_.mapKeys({ 'a': 1, 'b': 2 }, function(value, key) {
  return key + value;
});
// => { 'a1': 1, 'b2': 2 }
```

`_.mapValues(object, [iteratee=_.identity])`

Creates an object with the same keys as `object` and values generated by running each own enumerable string keyed property of `object` thru `iteratee`. The iteratee is invoked with three arguments: *(value, key, object)*.

Since

2.4.0

Arguments

`object` ***(Object)***: The object to iterate over.
`[iteratee=_.identity]` ***(Function)***: The function invoked per iteration.

Returns

(Object): Returns the new mapped object.

Example

```
var users = {
  'fred':    { 'user': 'fred',    'age': 40 },
  'pebbles': { 'user': 'pebbles', 'age': 1 }
};

_.mapValues(users, function(o) { return o.age; });
// => { 'fred': 40, 'pebbles': 1 } (iteration order is not guaranteed)

// The `_.property` iteratee shorthand.
_.mapValues(users, 'age');
// => { 'fred': 40, 'pebbles': 1 } (iteration order is not guaranteed)
```

_.merge(object, [sources])

This method is like `_.assign` except that it recursively merges own and inherited enumerable string keyed properties of source objects into the destination object. Source properties that resolve to `undefined` are skipped if a destination value exists. Array and plain object properties are merged recursively. Other objects and value types are overridden by assignment. Source objects are applied from left to right. Subsequent sources overwrite property assignments of previous sources.

Note: This method mutates `object`.

Since

0.5.0

Arguments

`object` ***(Object)***: The destination object.
`[sources]` ***(...Object)***: The source objects.

Returns

(Object): Returns `object`.

Example

```
var object = {
  'a': [{ 'b': 2 }, { 'd': 4 }]
};

var other = {
  'a': [{ 'c': 3 }, { 'e': 5 }]
};

_.merge(object, other);
// => { 'a': [{ 'b': 2, 'c': 3 }, { 'd': 4, 'e': 5 }] }
```

`_.mergeWith(object, sources, customizer)`

This method is like `_.merge` except that it accepts `customizer` which is invoked to produce the merged values of the destination and source properties. If `customizer` returns `undefined`, merging is handled by the method instead. The `customizer` is invoked with six arguments:
(objValue, srcValue, key, object, source, stack).

Note: This method mutates `object`.

Since

4.0.0

Arguments

`object` ***(Object)***: The destination object.
`sources` ***(...Object)***: The source objects.
`customizer` ***(Function)***: The function to customize assigned values.

Returns

(Object): Returns `object`.

Example

```
function customizer(objValue, srcValue) {
  if (_.isArray(objValue)) {
    return objValue.concat(srcValue);
  }
}

var object = { 'a': [1], 'b': [2] };
var other = { 'a': [3], 'b': [4] };

_.mergeWith(object, other, customizer);
// => { 'a': [1, 3], 'b': [2, 4] }
```

_.omit(object, [paths])

The opposite of `_.pick`; this method creates an object composed of the own and inherited enumerable property paths of `object` that are not omitted.

Note: This method is considerably slower than `_.pick`.

Since

0.1.0

Arguments

`object` ***(Object)***: The source object.
`[paths]` ***(...(string|string[]))***: The property paths to omit.

Returns

(Object): Returns the new object.

Example

```
var object = { 'a': 1, 'b': '2', 'c': 3 };

_.omit(object, ['a', 'c']);
// => { 'b': '2' }
```

.omitBy(object, [predicate=.identity])

The opposite of `_.pickBy`; this method creates an object composed of the own and inherited enumerable string keyed properties of `object` that `predicate` doesn't return truthy for. The predicate is invoked with two arguments: *(value, key)*.

Since

4.0.0

Arguments

`object` ***(Object)***: The source object.
`[predicate=_.identity]` ***(Function)***: The function invoked per property.

Returns

(Object): Returns the new object.

Example

```
var object = { 'a': 1, 'b': '2', 'c': 3 };

_.omitBy(object, _.isNumber);
// => { 'b': '2' }
```

`_.pick(object, [paths])`

Creates an object composed of the picked `object` properties.

Since

0.1.0

Arguments

`object` ***(Object)***: The source object.
`[paths]` ***(...(string|string[]))***: The property paths to pick.

Returns

(Object): Returns the new object.

Example

```
var object = { 'a': 1, 'b': '2', 'c': 3 };

_.pick(object, ['a', 'c']);
// => { 'a': 1, 'c': 3 }
```

`_.pickBy(object, [predicate=_.identity])`

Creates an object composed of the `object` properties `predicate` returns truthy for. The predicate is invoked with two arguments: *(value, key)*.

Since

4.0.0

Arguments

`object` ***(Object)***: The source object.
`[predicate=_.identity]` ***(Function)***: The function invoked per property.

Returns

(Object): Returns the new object.

Example

```
var object = { 'a': 1, 'b': '2', 'c': 3 };

_.pickBy(object, _.isNumber);
// => { 'a': 1, 'c': 3 }
```

`_.result(object, path, [defaultValue])`

This method is like `_.get` except that if the resolved value is a function it's invoked with the `this` binding of its parent object and its result is returned.

Since

0.1.0

Arguments

`object` ***(Object)***: The object to query.
`path` ***(Array|string)***: The path of the property to resolve.
`[defaultValue]` ***(*)***: The value returned for **`undefined`** resolved values.

Returns

(*): Returns the resolved value.

Example

```
var object = { 'a': [{ 'b': { 'c1': 3, 'c2': _.constant(4) } }] };

_.result(object, 'a[0].b.c1');
// => 3

_.result(object, 'a[0].b.c2');
// => 4

_.result(object, 'a[0].b.c3', 'default');
// => 'default'

_.result(object, 'a[0].b.c3', _.constant('default'));
// => 'default'
```

_.set(object, path, value)

Sets the value at `path` of `object`. If a portion of `path` doesn't exist, it's created. Arrays are created for missing index properties while objects are created for all other missing properties. Use `_.setWith` to customize `path` creation.

Note: This method mutates `object`.

Since

3.7.0

Arguments

`object` ***(Object)***: The object to modify.
`path` ***(Array|string)***: The path of the property to set.
`value` ***(*)***: The value to set.

Returns

(Object): Returns `object`.

Example

```
var object = { 'a': [{ 'b': { 'c': 3 } }] };

_.set(object, 'a[0].b.c', 4);
console.log(object.a[0].b.c);
// => 4

_.set(object, ['x', '0', 'y', 'z'], 5);
console.log(object.x[0].y.z);
// => 5
```

_.setWith(object, path, value, [customizer])

This method is like `_.set` except that it accepts `customizer` which is invoked to produce the objects of `path`. If `customizer` returns `undefined` path creation is handled by the method instead. The `customizer` is invoked with three arguments: *(nsValue, key, nsObject)*.

Note: This method mutates `object`.

Since

4.0.0

Arguments

`object` ***(Object)***: The object to modify.
`path` ***(Array|string)***: The path of the property to set.
`value` ***(*)***: The value to set.
`[customizer]` ***(Function)***: The function to customize assigned values.

Returns

(Object): Returns `object`.

Example

```
var object = {};

_.setWith(object, '[0][1]', 'a', Object);
// => { '0': { '1': 'a' } }
```

_.toPairs(object)

Creates an array of own enumerable string keyed-value pairs for `object` which can be consumed by `_.fromPairs`. If `object` is a map or set, its entries are returned.

Since

4.0.0

Aliases

_.entries

Arguments

object *(Object)*: The object to query.

Returns

(Array): Returns the key-value pairs.

Example

```
function Foo() {
  this.a = 1;
  this.b = 2;
}

Foo.prototype.c = 3;

_.toPairs(new Foo);
// => [['a', 1], ['b', 2]] (iteration order is not guaranteed)
```

`_.toPairsIn(object)`

Creates an array of own and inherited enumerable string keyed-value pairs for `object` which can be consumed by `_.fromPairs`. If `object` is a map or set, its entries are returned.

Since

4.0.0

Aliases

_.entriesIn

Arguments

object *(Object)*: The object to query.

Returns

(Array): Returns the key-value pairs.

Example

```
function Foo() {
  this.a = 1;
  this.b = 2;
}

Foo.prototype.c = 3;

_.toPairsIn(new Foo);
// => [['a', 1], ['b', 2], ['c', 3]] (iteration order is not guaranteed)
```

.transform(object, [iteratee=.identity], [accumulator])

An alternative to `_.reduce`; this method transforms `object` to a new `accumulator` object which is the result of running each of its own enumerable string keyed properties thru `iteratee`, with each invocation potentially mutating the `accumulator` object. If `accumulator` is not provided, a new object with the same `[[Prototype]]` will be used. The iteratee is invoked with four arguments: *(accumulator, value, key, object)*. Iteratee functions may exit iteration early by explicitly returning `false`.

Since

1.3.0

Arguments

`object` ***(Object)***: The object to iterate over.
`[iteratee=_.identity]` ***(Function)***: The function invoked per iteration.
`[accumulator]` ***(*)***: The custom accumulator value.

Returns

(*): Returns the accumulated value.

Example

```
_.transform([2, 3, 4], function(result, n) {
  result.push(n *= n);
  return n % 2 == 0;
}, []);
// => [4, 9]

_.transform({ 'a': 1, 'b': 2, 'c': 1 }, function(result, value, key) {
  (result[value] || (result[value] = [])).push(key);
}, {});
// => { '1': ['a', 'c'], '2': ['b'] }
```

_.unset(object, path)

Removes the property at `path` of `object`.

Note: This method mutates `object`.

Since

4.0.0

Arguments

`object` ***(Object)***: The object to modify.
`path` ***(Array|string)***: The path of the property to unset.

Returns

(boolean): Returns `true` if the property is deleted, else `false`.

Example

```
var object = { 'a': [{ 'b': { 'c': 7 } }] };
_.unset(object, 'a[0].b.c');
// => true

console.log(object);
// => { 'a': [{ 'b': {} }] };

_.unset(object, ['a', '0', 'b', 'c']);
// => true

console.log(object);
// => { 'a': [{ 'b': {} }] };
```

_.update(object, path, updater)

This method is like `_.set` except that accepts `updater` to produce the value to set. Use `_.updateWith` to customize `path` creation. The `updater` is invoked with one argument: *(value)*.

Note: This method mutates `object`.

Since

4.6.0

Arguments

`object` ***(Object)***: The object to modify.
`path` ***(Array|string)***: The path of the property to set.
`updater` ***(Function)***: The function to produce the updated value.

Returns

(Object): Returns `object`.

Example

```
var object = { 'a': [{ 'b': { 'c': 3 } }] };

_.update(object, 'a[0].b.c', function(n) { return n * n; });
console.log(object.a[0].b.c);
// => 9

_.update(object, 'x[0].y.z', function(n) { return n ? n + 1 : 0; });
console.log(object.x[0].y.z);
// => 0
```

`_.updateWith(object, path, updater, [customizer])`

This method is like `_.update` except that it accepts `customizer` which is invoked to produce the objects of `path`. If `customizer` returns `undefined` path creation is handled by the method instead. The `customizer` is invoked with three arguments: *(nsValue, key, nsObject)*.

Note: This method mutates `object`.

Since

4.6.0

Arguments

`object` ***(Object)***: The object to modify.
`path` ***(Array|string)***: The path of the property to set.
`updater` ***(Function)***: The function to produce the updated value.
`[customizer]` ***(Function)***: The function to customize assigned values.

Returns

(Object): Returns `object`.

Example

```
var object = {};

_.updateWith(object, '[0][1]', _.constant('a'), Object);
// => { '0': { '1': 'a' } }
```

_.values(object)

Creates an array of the own enumerable string keyed property values of `object`.

Note: Non-object values are coerced to objects.

Since

0.1.0

Arguments

object ***(Object)***: The object to query.

Returns

(Array): Returns the array of property values.

Example

```
function Foo() {
  this.a = 1;
  this.b = 2;
}

Foo.prototype.c = 3;

_.values(new Foo);
// => [1, 2] (iteration order is not guaranteed)

_.values('hi');
// => ['h', 'i']
```

_.valuesIn(object)

Creates an array of the own and inherited enumerable string keyed property values of `object`.

Note: Non-object values are coerced to objects.

Since

3.0.0

Arguments

object ***(Object)***: The object to query.

Returns

(Array): Returns the array of property values.

Example

```
function Foo() {
  this.a = 1;
  this.b = 2;
}

Foo.prototype.c = 3;

_.valuesIn(new Foo);
// => [1, 2, 3] (iteration order is not guaranteed)
```

“Seq” Methods

_(value)

Creates a `lodash` object which wraps `value` to enable implicit method chain sequences. Methods that operate on and return arrays, collections, and functions can be chained together. Methods that retrieve a single value or may return a primitive value will automatically end the chain sequence and return the unwrapped value. Otherwise, the value must be unwrapped with `_#value`.

Explicit chain sequences, which must be unwrapped with `_#value`, may be enabled using `_.chain`.

The execution of chained methods is lazy, that is, it's deferred until `_#value` is implicitly or explicitly called.

Lazy evaluation allows several methods to support shortcut fusion. Shortcut fusion is an optimization to merge iteratee calls; this avoids the creation of intermediate arrays and can greatly reduce the number of iteratee executions. Sections of a chain sequence qualify for shortcut fusion if the section is applied to an array and iteratees accept only one argument. The heuristic for whether a section qualifies for shortcut fusion is subject to change.

Chaining is supported in custom builds as long as the `_#value` method is directly or indirectly included in the build.

In addition to lodash methods, wrappers have `Array` and `String` methods.

The wrapper `Array` methods are:
`concat`, `join`, `pop`, `push`, `shift`, `sort`, `splice`, and `unshift`

The wrapper `String` methods are:
`replace` and `split`

The wrapper methods that support shortcut fusion are:
`at`, `compact`, `drop`, `dropRight`, `dropWhile`, `filter`, `find`, `findLast`, `head`, `initial`, `last`, `map`, `reject`, `reverse`, `slice`, `tail`, `take`, `takeRight`, `takeRightWhile`, `takeWhile`, and `toArray`

The chainable wrapper methods are:
`after`, `ary`, `assign`, `assignIn`, `assignInWith`, `assignWith`, `at`, `before`, `bind`, `bindAll`, `bindKey`, `castArray`, `chain`, `chunk`, `commit`, `compact`, `concat`, `conforms`, `constant`, `countBy`, `create`, `curry`, `debounce`, `defaults`, `defaultsDeep`, `defer`, `delay`, `difference`, `differenceBy`, `differenceWith`, `drop`, `dropRight`, `dropRightWhile`, `dropWhile`, `extend`, `extendWith`, `fill`, `filter`, `flatMap`, `flatMapDeep`, `flatMapDepth`, `flatten`, `flattenDeep`, `flattenDepth`, `flip`, `flow`, `flowRight`, `fromPairs`, `functions`, `functionsIn`, `groupBy`, `initial`, `intersection`, `intersectionBy`, `intersectionWith`, `invert`, `invertBy`, `invokeMap`, `iteratee`, `keyBy`, `keys`, `keysIn`, `map`, `mapKeys`, `mapValues`, `matches`, `matchesProperty`, `memoize`, `merge`, `mergeWith`, `method`, `methodOf`, `mixin`, `negate`, `nthArg`, `omit`, `omitBy`, `once`, `orderBy`, `over`, `overArgs`, `overEvery`, `overSome`, `partial`, `partialRight`, `partition`, `pick`, `pickBy`, `plant`, `property`, `propertyOf`, `pull`, `pullAll`, `pullAllBy`, `pullAllWith`, `pullAt`, `push`, `range`, `rangeRight`, `rearg`, `reject`, `remove`, `rest`, `reverse`, `sampleSize`, `set`, `setWith`, `shuffle`, `slice`, `sort`, `sortBy`, `sortedUniq`, `sortedUniqBy`, `splice`, `spread`, `tail`, `take`, `takeRight`, `takeRightWhile`, `takeWhile`, `tap`, `throttle`, `thru`, `toArray`, `toPairs`, `toPairsIn`, `toPath`, `toPlainObject`, `transform`, `unary`, `union`, `unionBy`, `unionWith`, `uniq`, `uniqBy`, `uniqWith`, `unset`, `unshift`, `unzip`, `unzipWith`, `update`, `updateWith`, `values`, `valuesIn`, `without`, `wrap`, `xor`, `xorBy`, `xorWith`, `zip`, `zipObject`, `zipObjectDeep`, and `zipWith`

The wrapper methods that are **not** chainable by default are:
`add`, `attempt`, `camelCase`, `capitalize`, `ceil`, `clamp`, `clone`, `cloneDeep`, `cloneDeepWith`, `cloneWith`, `conformsTo`, `deburr`, `defaultTo`, `divide`, `each`, `eachRight`, `endsWith`, `eq`, `escape`, `escapeRegExp`, `every`, `find`, `findIndex`, `findKey`, `findLast`, `findLastIndex`, `findLastKey`, `first`, `floor`, `forEach`, `forEachRight`, `forIn`, `forInRight`, `forOwn`, `forOwnRight`, `get`, `gt`, `gte`, `has`, `hasIn`, `head`, `identity`, `includes`, `indexOf`, `inRange`, `invoke`, `isArguments`, `isArray`, `isArrayBuffer`, `isArrayLike`, `isArrayLikeObject`, `isBoolean`, `isBuffer`, `isDate`, `isElement`, `isEmpty`, `isEqual`, `isEqualWith`, `isError`, `isFinite`, `isFunction`, `isInteger`, `isLength`, `isMap`, `isMatch`, `isMatchWith`, `isNaN`, `isNative`, `isNil`, `isNull`, `isNumber`, `isObject`, `isObjectLike`, `isPlainObject`, `isRegExp`, `isSafeInteger`, `isSet`, `isString`, `isUndefined`, `isTypedArray`, `isWeakMap`, `isWeakSet`, `join`, `kebabCase`, `last`, `lastIndexOf`, `lowerCase`, `lowerFirst`, `lt`, `lte`, `max`, `maxBy`, `mean`, `meanBy`, `min`, `minBy`, `multiply`, `noConflict`, `noop`, `now`, `nth`, `pad`, `padEnd`, `padStart`, `parseInt`, `pop`, `random`, `reduce`, `reduceRight`, `repeat`, `result`, `round`, `runInContext`, `sample`, `shift`, `size`, `snakeCase`, `some`, `sortedIndex`, `sortedIndexBy`, `sortedLastIndex`, `sortedLastIndexBy`, `startCase`, `startsWith`, `stubArray`, `stubFalse`, `stubObject`, `stubString`,

stubTrue, subtract, sum, sumBy, template, times, toFinite, toInteger, toJSON, toLength, toLower, toNumber, toSafeInteger, toString, toUpper, trim, trimEnd, trimStart, truncate, unescape, uniqueId, upperCase, upperFirst, value, and words

Arguments

value ***(*)***: The value to wrap in a **lodash** instance.

Returns

(Object): Returns the new lodash wrapper instance.

Example

```
function square(n) {
  return n * n;
}

var wrapped = _([1, 2, 3]);

// Returns an unwrapped value.
wrapped.reduce(_.add);
// => 6

// Returns a wrapped value.
var squares = wrapped.map(square);

_.isArray(squares);
// => false

_.isArray(squares.value());
// => true
```

_.chain(value)

Creates a lodash wrapper instance that wraps value with explicit method chain sequences enabled. The result of such sequences must be unwrapped with _#value.

Since

1.3.0

Arguments

value ***(*)***: The value to wrap.

Returns

(Object): Returns the new `lodash` wrapper instance.

Example

```
var users = [
  { 'user': 'barney',  'age': 36 },
  { 'user': 'fred',    'age': 40 },
  { 'user': 'pebbles', 'age': 1 }
];

var youngest = _
  .chain(users)
  .sortBy('age')
  .map(function(o) {
    return o.user + ' is ' + o.age;
  })
  .head()
  .value();
// => 'pebbles is 1'
```

_.tap(value, interceptor)

This method invokes `interceptor` and returns `value`. The interceptor is invoked with one argument; *(value)*. The purpose of this method is to "tap into" a method chain sequence in order to modify intermediate results.

Since

0.1.0

Arguments

`value` ***(*)***: The value to provide to **`interceptor`**.
`interceptor` ***(Function)***: The function to invoke.

Returns

(*): Returns `value`.

Example

```
_([1, 2, 3])
 .tap(function(array) {
// Mutate input array.
   array.pop();
 })
 .reverse()
 .value();
// => [2, 1]
```

_.thru(value, interceptor)

This method is like `_.tap` except that it returns the result of `interceptor`. The purpose of this method is to "pass thru" values replacing intermediate results in a method chain sequence.

Since

3.0.0

Arguments

`value` ***(*)***: The value to provide to **`interceptor`**.
`interceptor` ***(Function)***: The function to invoke.

Returns

(*): Returns the result of `interceptor`.

Example

```
_('  abc  ')
 .chain()
 .trim()
 .thru(function(value) {
   return [value];
 })
 .value();
// => ['abc']
```

_.prototype[Symbol.iterator]()

Enables the wrapper to be iterable.

Since

4.0.0

Returns

(Object): Returns the wrapper object.

Example

```
var wrapped = _([1, 2]);

wrapped[Symbol.iterator]() === wrapped;
// => true

Array.from(wrapped);
// => [1, 2]
```

_.prototype.at([paths])

This method is the wrapper version of `_.at`.

Since

1.0.0

Arguments

[paths] *(...(string|string[]))*: The property paths to pick.

Returns

(Object): Returns the new `lodash` wrapper instance.

Example

```
var object = { 'a': [{ 'b': { 'c': 3 } }, 4] };

_(object).at(['a[0].b.c', 'a[1]']).value();
// => [3, 4]
```

_.prototype.chain()

Creates a `lodash` wrapper instance with explicit method chain sequences enabled.

Since

0.1.0

Returns

(Object): Returns the new `lodash` wrapper instance.

Example

```
var users = [
  { 'user': 'barney', 'age': 36 },
  { 'user': 'fred',   'age': 40 }
];

// A sequence without explicit chaining.
_(users).head();
// => { 'user': 'barney', 'age': 36 }

// A sequence with explicit chaining.
_(users)
  .chain()
  .head()
  .pick('user')
  .value();
// => { 'user': 'barney' }
```

_.prototype.commit()

Executes the chain sequence and returns the wrapped result.

Since

3.2.0

Returns

(Object): Returns the new `lodash` wrapper instance.

Example

```
var array = [1, 2];
var wrapped = _(array).push(3);

console.log(array);
// => [1, 2]

wrapped = wrapped.commit();
console.log(array);
// => [1, 2, 3]

wrapped.last();
// => 3

console.log(array);
// => [1, 2, 3]
```

_.prototype.next()

Gets the next value on a wrapped object following the iterator protocol (https://mdn.io/iteration_protocols#iterator).

Since

4.0.0

Returns

(Object): Returns the next iterator value.

Example

```
var wrapped = _([1, 2]);

wrapped.next();
// => { 'done': false, 'value': 1 }

wrapped.next();
// => { 'done': false, 'value': 2 }

wrapped.next();
// => { 'done': true, 'value': undefined }
```

_.prototype.plant(value)

Creates a clone of the chain sequence planting `value` as the wrapped value.

Since

3.2.0

Arguments

`value` ***(*)***: The value to plant.

Returns

(Object): Returns the new `lodash` wrapper instance.

Example

```
function square(n) {
  return n * n;
}

var wrapped = _([1, 2]).map(square);
var other = wrapped.plant([3, 4]);

other.value();
// => [9, 16]

wrapped.value();
// => [1, 4]
```

`_.prototype.reverse()`

This method is the wrapper version of `_.reverse`.

Note: This method mutates the wrapped array.

Since

0.1.0

Returns

(Object): Returns the new `lodash` wrapper instance.

Example

```
var array = [1, 2, 3];

_(array).reverse().value()
// => [3, 2, 1]

console.log(array);
// => [3, 2, 1]
```

_.prototype.value()

Executes the chain sequence to resolve the unwrapped value.

Since

0.1.0

Aliases

_.prototype.toJSON, _.prototype.valueOf

Returns

(*): Returns the resolved unwrapped value.

Example

```
_([1, 2, 3]).value();
// => [1, 2, 3]
```

“String” Methods

_.camelCase([string=''])

Converts `string` to camel case (https://en.wikipedia.org/wiki/CamelCase).

Since

3.0.0

Arguments

[string=''] ***(string)***: The string to convert.

Returns

(string): Returns the camel cased string.

Example

```
_.camelCase('Foo Bar');
// => 'fooBar'

_.camelCase('--foo-bar--');
// => 'fooBar'

_.camelCase('__FOO_BAR__');
// => 'fooBar'
```

_.capitalize([string=''])

Converts the first character of `string` to upper case and the remaining to lower case.

Since

3.0.0

Arguments

[string=''] ***(string)***: The string to capitalize.

Returns

(string): Returns the capitalized string.

Example

```
_.capitalize('FRED');
// => 'Fred'
```

_.deburr([string=''])

Deburrs `string` by converting Latin-1 Supplement (https://en.wikipedia.org/wiki/Latin-1_Supplement_(Unicode_block)#Character_table) and Latin Extended-A (https://en.wikipedia.org/wiki/Latin_Extended-A) letters to basic Latin letters and removing combining diacritical marks (https://en.wikipedia.org/wiki/Combining_Diacritical_Marks).

Since

3.0.0

Arguments

[string=''] ***(string)***: The string to deburr.

Returns

(string): Returns the deburred string.

Example

```
_.deburr('déjà vu');
// => 'deja vu'
```

_.endsWith([string=''], [target], [position=string.length])

Checks if `string` ends with the given target string.

Since

3.0.0

Arguments

[string=''] *(string)*: The string to inspect.
[target] *(string)*: The string to search for.
[position=string.length] *(number)*: The position to search up to.

Returns

(boolean): Returns `true` if `string` ends with `target`, else `false`.

Example

```
_.endsWith('abc', 'c');
// => true

_.endsWith('abc', 'b');
// => false

_.endsWith('abc', 'b', 2);
// => true
```

_.escape([string=''])

Converts the characters "&", "<", ">", '"', and "'" in `string` to their corresponding HTML entities.

Note: No other characters are escaped. To escape additional characters use a third-party library like *he* (https://mths.be/he).

Though the ">" character is escaped for symmetry, characters like ">" and "/" don't need escaping in HTML and have no special meaning unless they're part of a tag or unquoted attribute value. See Mathias Bynens's article (https://mathiasbynens.be/notes/ambiguous-ampersands) *(under "semi-related fun fact")* for more details.

When working with HTML you should always quote attribute values (http://wonko.com/post/html-escaping) to reduce XSS vectors.

Since

0.1.0

Arguments

[string=''] ***(string)***: The string to escape.

Returns

(string): Returns the escaped string.

Example

```
_.escape('fred, barney, & pebbles');
// => 'fred, barney, &amp; pebbles'
```

_.escapeRegExp([string=''])

Escapes the `RegExp` special characters "^", "$", "", ".", "*", "+", "?", "(", ")", "[", "]", "{", "}", and "|" in `string`.

Since

3.0.0

Arguments

[string=''] ***(string)***: The string to escape.

Returns

(string): Returns the escaped string.

Example

```
_.escapeRegExp('[lodash](https://lodash.com/)');
// => '\[lodash\]\(https://lodash\.com/\)'
```

_.kebabCase([string=''])

Converts `string` to kebab case (https://en.wikipedia.org/wiki/Letter_case#Special_case_styles).

Since

3.0.0

Arguments

`[string='']` ***`(string)`***: The string to convert.

Returns

`(string)`: Returns the kebab cased string.

Example

```
_.kebabCase('Foo Bar');
// => 'foo-bar'

_.kebabCase('fooBar');
// => 'foo-bar'

_.kebabCase('__FOO_BAR__');
// => 'foo-bar'
```

`_.lowerCase([string=''])`

Converts `string`, as space separated words, to lower case.

Since

4.0.0

Arguments

`[string='']` ***`(string)`***: The string to convert.

Returns

`(string)`: Returns the lower cased string.

Example

```
_.lowerCase('--Foo-Bar--');
// => 'foo bar'

_.lowerCase('fooBar');
// => 'foo bar'

_.lowerCase('__FOO_BAR__');
// => 'foo bar'
```

_.lowerFirst([string=''])

Converts the first character of `string` to lower case.

Since

4.0.0

Arguments

[string=''] ***(string)***: The string to convert.

Returns

(string): Returns the converted string.

Example

```
_.lowerFirst('Fred');
// => 'fred'

_.lowerFirst('FRED');
// => 'fRED'
```

_.pad([string=''], [length=0], [chars=' '])

Pads `string` on the left and right sides if it's shorter than `length`. Padding characters are truncated if they can't be evenly divided by `length`.

Since

3.0.0

Arguments

[string=''] ***(string)***: The string to pad.

[length=0] ***(number)***: The padding length.
[chars=' '] ***(string)***: The string used as padding.

Returns

(string): Returns the padded string.

Example

```
_.pad('abc', 8);
// => '  abc   '

_.pad('abc', 8, '_-');
// => '_-abc_-_'

_.pad('abc', 3);
// => 'abc'
```

_.padEnd([string=''], [length=0], [chars=' '])

Pads `string` on the right side if it's shorter than `length`. Padding characters are truncated if they exceed `length`.

Since

4.0.0

Arguments

[string=''] ***(string)***: The string to pad.
[length=0] ***(number)***: The padding length.
[chars=' '] ***(string)***: The string used as padding.

Returns

(string): Returns the padded string.

Example

```
_.padEnd('abc', 6);
// => 'abc   '

_.padEnd('abc', 6, '_-');
// => 'abc_-_'

_.padEnd('abc', 3);
// => 'abc'
```

`_.padStart([string=''], [length=0], [chars=' '])`

Pads `string` on the left side if it's shorter than `length`. Padding characters are truncated if they exceed `length`.

Since

4.0.0

Arguments

`[string='']` ***(string)***: The string to pad.
`[length=0]` ***(number)***: The padding length.
`[chars=' ']` ***(string)***: The string used as padding.

Returns

(string): Returns the padded string.

Example

```
_.padStart('abc', 6);
// => '   abc'

_.padStart('abc', 6, '_-');
// => '_-_abc'

_.padStart('abc', 3);
// => 'abc'
```

`_.parseInt(string, [radix=10])`

Converts `string` to an integer of the specified radix. If `radix` is `undefined` or `0`, a `radix` of `10` is used unless `value` is a hexadecimal, in which case a `radix` of `16` is used.

Note: This method aligns with the ES5 implementation (https://es5.github.io/#x15.1.2.2) of `parseInt`.

Since

1.1.0

Arguments

`string` ***(string)***: The string to convert.
`[radix=10]` ***(number)***: The radix to interpret **`value`** by.

Returns

(number): Returns the converted integer.

Example

```
_.parseInt('08');
// => 8

_.map(['6', '08', '10'], _.parseInt);
// => [6, 8, 10]
```

_.repeat([string=''], [n=1])

Repeats the given string n times.

Since

3.0.0

Arguments

[string=''] *(string)*: The string to repeat.
[n=1] *(number)*: The number of times to repeat the string.

Returns

(string): Returns the repeated string.

Example

```
_.repeat('*', 3);
// => '***'

_.repeat('abc', 2);
// => 'abcabc'

_.repeat('abc', 0);
// => ''
```

_.replace([string=''], pattern, replacement)

Replaces matches for `pattern` in `string` with `replacement`.

Note: This method is based on `String#replace` (https://mdn.io/String/replace).

Since

4.0.0

Arguments

`[string='']` *`(string)`*: The string to modify.
`pattern` *`(RegExp|string)`*: The pattern to replace.
`replacement` *`(Function|string)`*: The match replacement.

Returns

`(string)`: Returns the modified string.

Example

```
_.replace('Hi Fred', 'Fred', 'Barney');
// => 'Hi Barney'
```

`_.snakeCase([string=''])`

Converts `string` to snake case (https://en.wikipedia.org/wiki/Snake_case).

Since

3.0.0

Arguments

`[string='']` *`(string)`*: The string to convert.

Returns

`(string)`: Returns the snake cased string.

Example

```
_.snakeCase('Foo Bar');
// => 'foo_bar'

_.snakeCase('fooBar');
// => 'foo_bar'

_.snakeCase('--FOO-BAR--');
// => 'foo_bar'
```

`_.split([string=''], separator, [limit])`

Splits `string` by `separator`.

Note: This method is based on `String#split` (https://mdn.io/String/split).

Since

4.0.0

Arguments

`[string='']` *`(string)`*: The string to split.
`separator` *`(RegExp|string)`*: The separator pattern to split by.
`[limit]` *`(number)`*: The length to truncate results to.

Returns

`(Array)`: Returns the string segments.

Example

```
_.split('a-b-c', '-', 2);
// => ['a', 'b']
```

`_.startCase([string=''])`

Converts `string` to start case (https://en.wikipedia.org/wiki/Letter_case#Stylistic_or_specialised_usage).

Since

3.1.0

Arguments

`[string='']` *`(string)`*: The string to convert.

Returns

`(string)`: Returns the start cased string.

Example

```
_.startCase('--foo-bar--');
// => 'Foo Bar'

_.startCase('fooBar');
// => 'Foo Bar'

_.startCase('__FOO_BAR__');
// => 'FOO BAR'
```

_.startsWith([string=''], [target], [position=0])

Checks if `string` starts with the given target string.

Since

3.0.0

Arguments

[string=''] ***(string)***: The string to inspect.
[target] ***(string)***: The string to search for.
[position=0] ***(number)***: The position to search from.

Returns

(boolean): Returns `true` if `string` starts with `target`, else `false`.

Example

```
_.startsWith('abc', 'a');
// => true

_.startsWith('abc', 'b');
// => false

_.startsWith('abc', 'b', 1);
// => true
```

_.template([string=''], [options={}])

Creates a compiled template function that can interpolate data properties in "interpolate" delimiters, HTML-escape interpolated data properties in "escape" delimiters, and execute JavaScript in "evaluate" delimiters. Data properties may be accessed as free variables in the template. If a setting object is given, it takes precedence over `_.templateSettings` values.

Note: In the development build `_.template` utilizes sourceURLs (http://www.html5rocks.com/en/tutorials/developertools/sourcemaps/#toc-sourceurl) for easier debugging.

For more information on precompiling templates see lodash's custom builds documentation (https://lodash.com/custom-builds).

For more information on Chrome extension sandboxes see Chrome's extensions documentation (https://developer.chrome.com/extensions/sandboxingEval).

Since

0.1.0

Arguments

[string=''] ***(string)***: The template string.
[options={}] ***(Object)***: The options object.
[options.escape=_.templateSettings.escape] ***(RegExp)***: The HTML "escape" delimiter.
[options.evaluate=_.templateSettings.evaluate] ***(RegExp)***: The "evaluate" delimiter.
[options.imports=_.templateSettings.imports] ***(Object)***: An object to import into the template as free variables.
[options.interpolate=_.templateSettings.interpolate] ***(RegExp)***: The "interpolate" delimiter.
[options.sourceURL='lodash.templateSources[n]'] ***(string)***: The sourceURL of the compiled template.
[options.variable='obj'] ***(string)***: The data object variable name.

Returns

(Function): Returns the compiled template function.

Example

```
// Use the "interpolate" delimiter to create a compiled template.
var compiled = _.template('hello <%= user %>!');
compiled({ 'user': 'fred' });
// => 'hello fred!'

// Use the HTML "escape" delimiter to escape data property values.
var compiled = _.template('<b><%- value %></b>');
compiled({ 'value': '<script>' });
// => '<b>&lt;script&gt;</b>'

// Use the "evaluate" delimiter to execute JavaScript and generate HTML.
var compiled = _.template('<% _.forEach(users, function(user) { %><li><%- user %>
</li><% }); %>');
compiled({ 'users': ['fred', 'barney'] });
// => '<li>fred</li><li>barney</li>'

// Use the internal `print` function in "evaluate" delimiters.
var compiled = _.template('<% print("hello " + user); %>!');
compiled({ 'user': 'barney' });
// => 'hello barney!'

// Use the ES template literal delimiter as an "interpolate" delimiter.
// Disable support by replacing the "interpolate" delimiter.
var compiled = _.template('hello ${ user }!');
compiled({ 'user': 'pebbles' });
// => 'hello pebbles!'

// Use backslashes to treat delimiters as plain text.
var compiled = _.template('<%= "\\<%- value %\\>" %>');
compiled({ 'value': 'ignored' });
// => '<%- value %>'

// Use the `imports` option to import `jQuery` as `jq`.
var text = '<% jq.each(users, function(user) { %><li><%- user %></li><% }); %>';
var compiled = _.template(text, { 'imports': { 'jq': jQuery } });
compiled({ 'users': ['fred', 'barney'] });
// => '<li>fred</li><li>barney</li>'

// Use the `sourceURL` option to specify a custom sourceURL for the template.
var compiled = _.template('hello <%= user %>!', { 'sourceURL': '/basic/greeting.jst'
compiled(data);
// => Find the source of "greeting.jst" under the Sources tab or Resources panel of t

// Use the `variable` option to ensure a with-
statement isn't used in the compiled template.
var compiled = _.template('hi <%= data.user %>!', { 'variable': 'data' });
```

```
compiled.source;
// => function(data) {
//   var __t, __p = '';
//   __p += 'hi ' + ((__t = ( data.user )) == null ? '' : __t) + '!';
//   return __p;
// }

// Use custom template delimiters.
_.templateSettings.interpolate = /{{([\s\S]+?)}}/g;
var compiled = _.template('hello {{ user }}!');
compiled({ 'user': 'mustache' });
// => 'hello mustache!'

// Use the `source` property to inline compiled templates for meaningful
// line numbers in error messages and stack traces.
fs.writeFileSync(path.join(process.cwd(), 'jst.js'), '\
  var JST = {\
    "main": ' + _.template(mainText).source + '\
  };\
');
```

_.toLower([string=''])

Converts `string`, as a whole, to lower case just like String#toLowerCase (https://mdn.io/toLowerCase).

Since

4.0.0

Arguments

[string=''] ***(string)***: The string to convert.

Returns

(string): Returns the lower cased string.

Example

```
_.toLower('--Foo-Bar--');
// => '--foo-bar--'

_.toLower('fooBar');
// => 'foobar'

_.toLower('__FOO_BAR__');
// => '__foo_bar__'
```

_.toUpper([string=''])

Converts `string`, as a whole, to upper case just like String#toUpperCase (https://mdn.io/toUpperCase).

Since

4.0.0

Arguments

[string=''] ***(string)***: The string to convert.

Returns

(string): Returns the upper cased string.

Example

```
_.toUpper('--foo-bar--');
// => '--FOO-BAR--'

_.toUpper('fooBar');
// => 'FOOBAR'

_.toUpper('__foo_bar__');
// => '__FOO_BAR__'
```

_.trim([string=''], [chars=whitespace])

Removes leading and trailing whitespace or specified characters from `string`.

Since

3.0.0

Arguments

[string=''] ***(string)***: The string to trim.
[chars=whitespace] ***(string)***: The characters to trim.

Returns

(string): Returns the trimmed string.

Example

```
_.trim('  abc  ');
// => 'abc'

_.trim('-_-abc-_-', '_-');
// => 'abc'

_.map(['  foo  ', '  bar  '], _.trim);
// => ['foo', 'bar']
```

_.trimEnd([string=''], [chars=whitespace])

Removes trailing whitespace or specified characters from `string`.

Since

4.0.0

Arguments

[string=''] ***(string)***: The string to trim.
[chars=whitespace] ***(string)***: The characters to trim.

Returns

(string): Returns the trimmed string.

Example

```
_.trimEnd('  abc  ');
// => '  abc'

_.trimEnd('-_-abc-_-', '_-');
// => '-_-abc'
```

_.trimStart([string=''], [chars=whitespace])

Removes leading whitespace or specified characters from `string`.

Since

4.0.0

Arguments

[string=''] ***(string)***: The string to trim.
[chars=whitespace] ***(string)***: The characters to trim.

Returns

(string): Returns the trimmed string.

Example

```
_.trimStart('  abc  ');
// => 'abc  '

_.trimStart('-_-abc-_-', '_-');
// => 'abc-_-'
```

`_.truncate([string=''], [options={}])`

Truncates `string` if it's longer than the given maximum string length. The last characters of the truncated string are replaced with the omission string which defaults to "...".

Since

4.0.0

Arguments

[string=''] ***(string)***: The string to truncate.
[options={}] ***(Object)***: The options object.
[options.length=30] ***(number)***: The maximum string length.
[options.omission='...'] ***(string)***: The string to indicate text is omitted.
[options.separator] ***(RegExp|string)***: The separator pattern to truncate to.

Returns

(string): Returns the truncated string.

Example

```
_.truncate('hi-diddly-ho there, neighborino');
// => 'hi-diddly-ho there, neighbo...'

_.truncate('hi-diddly-ho there, neighborino', {
  'length': 24,
  'separator': ' '
});
// => 'hi-diddly-ho there,...'

_.truncate('hi-diddly-ho there, neighborino', {
  'length': 24,
  'separator': /,? +/
});
// => 'hi-diddly-ho there...'

_.truncate('hi-diddly-ho there, neighborino', {
  'omission': ' [...]'
});
// => 'hi-diddly-ho there, neig [...]'
```

_.unescape([string=''])

The inverse of `_.escape`; this method converts the HTML entities `&`, `<`, `>`, `&quot;`, and `&#39;` in `string` to their corresponding characters.

Note: No other HTML entities are unescaped. To unescape additional HTML entities use a third-party library like *he* (https://mths.be/he).

Since

0.6.0

Arguments

[string=''] ***(string)***: The string to unescape.

Returns

(string): Returns the unescaped string.

Example

```
_.unescape('fred, barney, &amp; pebbles');
// => 'fred, barney, & pebbles'
```

`_.upperCase([string=''])`

Converts `string`, as space separated words, to upper case.

Since

4.0.0

Arguments

`[string='']` ***`(string)`***: The string to convert.

Returns

`(string)`: Returns the upper cased string.

Example

```
_.upperCase('--foo-bar');
// => 'FOO BAR'

_.upperCase('fooBar');
// => 'FOO BAR'

_.upperCase('__foo_bar__');
// => 'FOO BAR'
```

`_.upperFirst([string=''])`

Converts the first character of `string` to upper case.

Since

4.0.0

Arguments

`[string='']` ***`(string)`***: The string to convert.

Returns

`(string)`: Returns the converted string.

Example

```
_.upperFirst('fred');
// => 'Fred'

_.upperFirst('FRED');
// => 'FRED'
```

_.words([string=''], [pattern])

Splits `string` into an array of its words.

Since

3.0.0

Arguments

[string=''] ***(string)***: The string to inspect.
[pattern] ***(RegExp|string)***: The pattern to match words.

Returns

(Array): Returns the words of `string`.

Example

```
_.words('fred, barney, & pebbles');
// => ['fred', 'barney', 'pebbles']

_.words('fred, barney, & pebbles', /[^, ]+/g);
// => ['fred', 'barney', '&', 'pebbles']
```

“Util” Methods

_.attempt(func, [args])

Attempts to invoke `func`, returning either the result or the caught error object. Any additional arguments are provided to `func` when it's invoked.

Since

3.0.0

Arguments

func ***(Function)***: The function to attempt.

`[args]` *`(...*)`*: The arguments to invoke **`func`** with.

Returns

`(*)`: Returns the `func` result or error object.

Example

```
// Avoid throwing errors for invalid selectors.
var elements = _.attempt(function(selector) {
  return document.querySelectorAll(selector);
}, '>_>');

if (_.isError(elements)) {
  elements = [];
}
```

`_.bindAll(object, methodNames)`

Binds methods of an object to the object itself, overwriting the existing method.

Note: This method doesn't set the "length" property of bound functions.

Since

0.1.0

Arguments

`object` ***`(Object)`***: The object to bind and assign the bound methods to.
`methodNames` ***`(...(string|string[]))`***: The object method names to bind.

Returns

`(Object)`: Returns `object`.

Example

```
var view = {
  'label': 'docs',
  'click': function() {
    console.log('clicked ' + this.label);
  }
};

_.bindAll(view, ['click']);
jQuery(element).on('click', view.click);
// => Logs 'clicked docs' when clicked.
```

_.cond(pairs)

Creates a function that iterates over `pairs` and invokes the corresponding function of the first predicate to return truthy. The predicate-function pairs are invoked with the `this` binding and arguments of the created function.

Since

4.0.0

Arguments

`pairs` ***(Array)***: The predicate-function pairs.

Returns

(Function): Returns the new composite function.

Example

```
var func = _.cond([
  [_.matches({ 'a': 1 }),           _.constant('matches A')],
  [_.conforms({ 'b': _.isNumber }), _.constant('matches B')],
  [_.stubTrue,                      _.constant('no match')]
]);

func({ 'a': 1, 'b': 2 });
// => 'matches A'

func({ 'a': 0, 'b': 1 });
// => 'matches B'

func({ 'a': '1', 'b': '2' });
// => 'no match'
```

_.conforms(source)

Creates a function that invokes the predicate properties of `source` with the corresponding property values of a given object, returning `true` if all predicates return truthy, else `false`.

Note: The created function is equivalent to `_.conformsTo` with `source` partially applied.

Since

4.0.0

Arguments

source ***(Object)***: The object of property predicates to conform to.

Returns

(Function): Returns the new spec function.

Example

```
var objects = [
  { 'a': 2, 'b': 1 },
  { 'a': 1, 'b': 2 }
];

_.filter(objects, _.conforms({ 'b': function(n) { return n > 1; } }));
// => [{ 'a': 1, 'b': 2 }]
```

_.constant(value)

Creates a function that returns `value`.

Since

2.4.0

Arguments

value ***(*)***: The value to return from the new function.

Returns

(Function): Returns the new constant function.

Example

```
var objects = _.times(2, _.constant({ 'a': 1 }));

console.log(objects);
// => [{ 'a': 1 }, { 'a': 1 }]

console.log(objects[0] === objects[1]);
// => true
```

_.defaultTo(value, defaultValue)

Checks `value` to determine whether a default value should be returned in its place. The `defaultValue` is returned if `value` is `NaN`, `null`, or `undefined`.

Since

4.14.0

Arguments

`value` ***(*)***: The value to check.
`defaultValue` ***(*)***: The default value.

Returns

(*): Returns the resolved value.

Example

```
_.defaultTo(1, 10);
// => 1

_.defaultTo(undefined, 10);
// => 10
```

_.flow([funcs])

Creates a function that returns the result of invoking the given functions with the `this` binding of the created function, where each successive invocation is supplied the return value of the previous.

Since

3.0.0

Arguments

`[funcs]` ***(...(Function|Function[]))***: The functions to invoke.

Returns

(Function): Returns the new composite function.

Example

```
function square(n) {
  return n * n;
}

var addSquare = _.flow([_.add, square]);
addSquare(1, 2);
// => 9
```

_.flowRight([funcs])

This method is like `_.flow` except that it creates a function that invokes the given functions from right to left.

Since

3.0.0

Arguments

[funcs] ***(...(Function|Function[]))***: The functions to invoke.

Returns

(Function): Returns the new composite function.

Example

```
function square(n) {
  return n * n;
}

var addSquare = _.flowRight([square, _.add]);
addSquare(1, 2);
// => 9
```

_.identity(value)

This method returns the first argument it receives.

Since

0.1.0

Arguments

`value (*)`: Any value.

Returns

(*): Returns `value`.

Example

```
var object = { 'a': 1 };

console.log(_.identity(object) === object);
// => true
```

`_.iteratee([func=_.identity])`

Creates a function that invokes `func` with the arguments of the created function. If `func` is a property name, the created function returns the property value for a given element. If `func` is an array or object, the created function returns `true` for elements that contain the equivalent source properties, otherwise it returns `false`.

Since

4.0.0

Arguments

`[func=_.identity]` ***(*)***: The value to convert to a callback.

Returns

(Function): Returns the callback.

Example

```
var users = [
  { 'user': 'barney', 'age': 36, 'active': true },
  { 'user': 'fred',   'age': 40, 'active': false }
];

// The `_.matches` iteratee shorthand.
_.filter(users, _.iteratee({ 'user': 'barney', 'active': true }));
// => [{ 'user': 'barney', 'age': 36, 'active': true }]

// The `_.matchesProperty` iteratee shorthand.
_.filter(users, _.iteratee(['user', 'fred']));
// => [{ 'user': 'fred', 'age': 40 }]

// The `_.property` iteratee shorthand.
_.map(users, _.iteratee('user'));
// => ['barney', 'fred']

// Create custom iteratee shorthands.
_.iteratee = _.wrap(_.iteratee, function(iteratee, func) {
  return !_.isRegExp(func) ? iteratee(func) : function(string) {
    return func.test(string);
  };
});

_.filter(['abc', 'def'], /ef/);
// => ['def']
```

`_.matches(source)`

Creates a function that performs a partial deep comparison between a given object and `source`, returning `true` if the given object has equivalent property values, else `false`.

Note: The created function is equivalent to `_.isMatch` with `source` partially applied.

Partial comparisons will match empty array and empty object `source` values against any array or object value, respectively. See `_.isEqual` for a list of supported value comparisons.

Since

3.0.0

Arguments

`source` ***(Object)***: The object of property values to match.

Returns

(Function): Returns the new spec function.

Example

```
var objects = [
  { 'a': 1, 'b': 2, 'c': 3 },
  { 'a': 4, 'b': 5, 'c': 6 }
];

_.filter(objects, _.matches({ 'a': 4, 'c': 6 }));
// => [{ 'a': 4, 'b': 5, 'c': 6 }]
```

`_.matchesProperty(path, srcValue)`

Creates a function that performs a partial deep comparison between the value at `path` of a given object to `srcValue`, returning `true` if the object value is equivalent, else `false`.

Note: Partial comparisons will match empty array and empty object `srcValue` values against any array or object value, respectively. See `_.isEqual` for a list of supported value comparisons.

Since

3.2.0

Arguments

path *(Array|string)*: The path of the property to get.
srcValue *(*)*: The value to match.

Returns

(Function): Returns the new spec function.

Example

```
var objects = [
  { 'a': 1, 'b': 2, 'c': 3 },
  { 'a': 4, 'b': 5, 'c': 6 }
];

_.find(objects, _.matchesProperty('a', 4));
// => { 'a': 4, 'b': 5, 'c': 6 }
```

_.method(path, [args])

Creates a function that invokes the method at `path` of a given object. Any additional arguments are provided to the invoked method.

Since

3.7.0

Arguments

`path` ***(Array|string)***: The path of the method to invoke.
`[args]` ***(...*)***: The arguments to invoke the method with.

Returns

(Function): Returns the new invoker function.

Example

```
var objects = [
  { 'a': { 'b': _.constant(2) } },
  { 'a': { 'b': _.constant(1) } }
];

_.map(objects, _.method('a.b'));
// => [2, 1]

_.map(objects, _.method(['a', 'b']));
// => [2, 1]
```

_.methodOf(object, [args])

The opposite of `_.method`; this method creates a function that invokes the method at a given path of `object`. Any additional arguments are provided to the invoked method.

Since

3.7.0

Arguments

`object` ***(Object)***: The object to query.
`[args]` ***(...*)***: The arguments to invoke the method with.

Returns

(Function): Returns the new invoker function.

Example

```
var array = _.times(3, _.constant),
    object = { 'a': array, 'b': array, 'c': array };

_.map(['a[2]', 'c[0]'], _.methodOf(object));
// => [2, 0]

_.map([['a', '2'], ['c', '0']], _.methodOf(object));
// => [2, 0]
```

`_.mixin([object=lodash], source, [options={}])`

Adds all own enumerable string keyed function properties of a source object to the destination object. If `object` is a function, then methods are added to its prototype as well.

Note: Use `_.runInContext` to create a pristine `lodash` function to avoid conflicts caused by modifying the original.

Since

0.1.0

Arguments

`[object=lodash]` ***(Function|Object)***: The destination object.
`source` ***(Object)***: The object of functions to add.
`[options={}]` ***(Object)***: The options object.
`[options.chain=true]` ***(boolean)***: Specify whether mixins are chainable.

Returns

(*): Returns `object`.

Example

```
function vowels(string) {
  return _.filter(string, function(v) {
    return /[aeiou]/i.test(v);
  });
}

_.mixin({ 'vowels': vowels });
_.vowels('fred');
// => ['e']

_('fred').vowels().value();
// => ['e']

_.mixin({ 'vowels': vowels }, { 'chain': false });
_('fred').vowels();
// => ['e']
```

_.noConflict()

Reverts the _ variable to its previous value and returns a reference to the `lodash` function.

Since

0.1.0

Returns

(Function): Returns the `lodash` function.

Example

```
var lodash = _.noConflict();
```

_.noop()

This method returns `undefined`.

Since

2.3.0

Example

```
_.times(2, _.noop);
// => [undefined, undefined]
```

_.nthArg([n=0])

Creates a function that gets the argument at index n. If n is negative, the nth argument from the end is returned.

Since

4.0.0

Arguments

[n=0] ***(number)***: The index of the argument to return.

Returns

(Function): Returns the new pass-thru function.

Example

```
var func = _.nthArg(1);
func('a', 'b', 'c', 'd');
// => 'b'

var func = _.nthArg(-2);
func('a', 'b', 'c', 'd');
// => 'c'
```

.over([iteratees=[.identity]])

Creates a function that invokes `iteratees` with the arguments it receives and returns their results.

Since

4.0.0

Arguments

[iteratees=[_.identity]] ***(...(Function|Function[]))***: The iteratees to invoke.

Returns

(Function): Returns the new function.

Example

```
var func = _.over([Math.max, Math.min]);

func(1, 2, 3, 4);
// => [4, 1]
```

.overEvery([predicates=[.identity]])

Creates a function that checks if **all** of the `predicates` return truthy when invoked with the arguments it receives.

Since

4.0.0

Arguments

`[predicates=[_.identity]]` ***(...(Function|Function[]))***: The predicates to check.

Returns

(Function): Returns the new function.

Example

```
var func = _.overEvery([Boolean, isFinite]);

func('1');
// => true

func(null);
// => false

func(NaN);
// => false
```

.overSome([predicates=[.identity]])

Creates a function that checks if **any** of the `predicates` return truthy when invoked with the arguments it receives.

Since

4.0.0

Arguments

[predicates=[_.identity]] ***(...(Function|Function[]))***: The predicates to check.

Returns

(Function): Returns the new function.

Example

```
var func = _.overSome([Boolean, isFinite]);

func('1');
// => true

func(null);
// => true

func(NaN);
// => false
```

_.property(path)

Creates a function that returns the value at `path` of a given object.

Since

2.4.0

Arguments

path ***(Array|string)***: The path of the property to get.

Returns

(Function): Returns the new accessor function.

Example

```
var objects = [
  { 'a': { 'b': 2 } },
  { 'a': { 'b': 1 } }
];

_.map(objects, _.property('a.b'));
// => [2, 1]

_.map(_.sortBy(objects, _.property(['a', 'b'])), 'a.b');
// => [1, 2]
```

_.propertyOf(object)

The opposite of `_.property`; this method creates a function that returns the value at a given path of `object`.

Since

3.0.0

Arguments

`object` ***(Object)***: The object to query.

Returns

(Function): Returns the new accessor function.

Example

```
var array = [0, 1, 2],
    object = { 'a': array, 'b': array, 'c': array };

_.map(['a[2]', 'c[0]'], _.propertyOf(object));
// => [2, 0]

_.map([['a', '2'], ['c', '0']], _.propertyOf(object));
// => [2, 0]
```

_.range([start=0], end, [step=1])

Creates an array of numbers *(positive and/or negative)* progressing from `start` up to, but not including, `end`. A step of `-1` is used if a negative `start` is specified without an `end` or `step`. If `end` is not specified, it's set to `start` with `start` then set to `0`.

Note: JavaScript follows the IEEE-754 standard for resolving floating-point values which can produce unexpected results.

Since

0.1.0

Arguments

`[start=0]` ***(number)***: The start of the range.
`end` ***(number)***: The end of the range.

[step=1] ***(number)***: The value to increment or decrement by.

Returns

(Array): Returns the range of numbers.

Example

```
_.range(4);
// => [0, 1, 2, 3]

_.range(-4);
// => [0, -1, -2, -3]

_.range(1, 5);
// => [1, 2, 3, 4]

_.range(0, 20, 5);
// => [0, 5, 10, 15]

_.range(0, -4, -1);
// => [0, -1, -2, -3]

_.range(1, 4, 0);
// => [1, 1, 1]

_.range(0);
// => []
```

_.rangeRight([start=0], end, [step=1])

This method is like `_.range` except that it populates values in descending order.

Since

4.0.0

Arguments

[start=0] ***(number)***: The start of the range.
end ***(number)***: The end of the range.
[step=1] ***(number)***: The value to increment or decrement by.

Returns

(Array): Returns the range of numbers.

Example

```
_.rangeRight(4);
// => [3, 2, 1, 0]

_.rangeRight(-4);
// => [-3, -2, -1, 0]

_.rangeRight(1, 5);
// => [4, 3, 2, 1]

_.rangeRight(0, 20, 5);
// => [15, 10, 5, 0]

_.rangeRight(0, -4, -1);
// => [-3, -2, -1, 0]

_.rangeRight(1, 4, 0);
// => [1, 1, 1]

_.rangeRight(0);
// => []
```

`_.runInContext([context=root])`

Create a new pristine `lodash` function using the `context` object.

Since

1.1.0

Arguments

`[context=root]` ***(Object)***: The context object.

Returns

(Function): Returns a new `lodash` function.

Example

```
_.mixin({ 'foo': _.constant('foo') });

var lodash = _.runInContext();
lodash.mixin({ 'bar': lodash.constant('bar') });

_.isFunction(_.foo);
// => true
_.isFunction(_.bar);
// => false

lodash.isFunction(lodash.foo);
// => false
lodash.isFunction(lodash.bar);
// => true

// Create a suped-up `defer` in Node.js.
var defer = _.runInContext({ 'setTimeout': setImmediate }).defer;
```

_.stubArray()

This method returns a new empty array.

Since

4.13.0

Returns

(Array): Returns the new empty array.

Example

```
var arrays = _.times(2, _.stubArray);

console.log(arrays);
// => [[], []]

console.log(arrays[0] === arrays[1]);
// => false
```

_.stubFalse()

This method returns `false`.

Since

4.13.0

Returns

(boolean): Returns `false`.

Example

```
_.times(2, _.stubFalse);
// => [false, false]
```

_.stubObject()

This method returns a new empty object.

Since

4.13.0

Returns

(Object): Returns the new empty object.

Example

```
var objects = _.times(2, _.stubObject);

console.log(objects);
// => [{}, {}]

console.log(objects[0] === objects[1]);
// => false
```

_.stubString()

This method returns an empty string.

Since

4.13.0

Returns

(string): Returns the empty string.

Example

```
_.times(2, _.stubString);
// => ['', '']
```

_.stubTrue()

This method returns `true`.

Since

4.13.0

Returns

(boolean): Returns `true`.

Example

```
_.times(2, _.stubTrue);
// => [true, true]
```

.times(n, [iteratee=.identity])

Invokes the iteratee `n` times, returning an array of the results of each invocation. The iteratee is invoked with one argument; *(index)*.

Since

0.1.0

Arguments

`n` ***(number)***: The number of times to invoke **`iteratee`**.
`[iteratee=_.identity]` ***(Function)***: The function invoked per iteration.

Returns

(Array): Returns the array of results.

Example

```
_.times(3, String);
// => ['0', '1', '2']

 _.times(4, _.constant(0));
// => [0, 0, 0, 0]
```

_.toPath(value)

Converts `value` to a property path array.

Since

4.0.0

Arguments

`value` ***(*)***: The value to convert.

Returns

(Array): Returns the new property path array.

Example

```
_.toPath('a.b.c');
// => ['a', 'b', 'c']

_.toPath('a[0].b.c');
// => ['a', '0', 'b', 'c']
```

_.uniqueId([prefix=''])

Generates a unique ID. If `prefix` is given, the ID is appended to it.

Since

0.1.0

Arguments

`[prefix='']` ***(string)***: The value to prefix the ID with.

Returns

(string): Returns the unique ID.

Example

```
_.uniqueId('contact_');
// => 'contact_104'

_.uniqueId();
// => '105'
```

Properties

_.VERSION

(string): The semantic version number.

_.templateSettings

(Object): By default, the template delimiters used by lodash are like those in embedded Ruby *(ERB)* as well as ES2015 template strings. Change the following template settings to use alternative delimiters.

_.templateSettings.escape

(RegExp): Used to detect `data` property values to be HTML-escaped.

_.templateSettings.evaluate

(RegExp): Used to detect code to be evaluated.

_.templateSettings.imports

(Object): Used to import variables into the compiled template.

`_.templateSettings.interpolate`

(RegExp): Used to detect `data` property values to inject.

`_.templateSettings.variable`

(string): Used to reference the data object in the template text.

Methods

`_.templateSettings.imports._`

A reference to the `lodash` function.