* [Introduction](#gjdgxs)
  + [Places](#30j0zll)
  + [Why underscore-contrib?](#1fob9te)
  + [Use](#3znysh7)
    - [In the Browser](#2et92p0)
    - [In Node.js](#tyjcwt)
  + [License](#3dy6vkm)
* [Sub-libraries](#1t3h5sf)
  + [array.builders](#4d34og8)
    - [cat](#2s8eyo1)
    - [chunk](#17dp8vu)
    - [chunkAll](#3rdcrjn)
    - [cons](#26in1rg)
    - [cycle](#lnxbz9)
    - [interpose](#35nkun2)
    - [iterateUntil](#1ksv4uv)
    - [keepIndexed](#44sinio)
    - [mapcat](#2jxsxqh)
    - [reductions](#z337ya)
    - [repeat](#3j2qqm3)
    - [splitAt](#1y810tw)
    - [takeSkipping](#4i7ojhp)
    - [weave](#2xcytpi)
  + [array.selectors](#1ci93xb)
    - [best](#3whwml4)
    - [dropWhile](#2bn6wsx)
    - [keep](#qsh70q)
    - [nth](#3as4poj)
    - [partitionBy](#1pxezwc)
    - [second](#49x2ik5)
    - [takeWhile](#2p2csry)
    - [third](#147n2zr)
  + [collections.walk](#3o7alnk)
  + [function.arity](#23ckvvd)
    - [arity](#ihv636)
    - [binary](#32hioqz)
    - [curry](#1hmsyys)
    - [curry2](#41mghml)
    - [curry3](#2grqrue)
    - [fix](#vx1227)
    - [quaternary](#3fwokq0)
    - [ternary](#1v1yuxt)
    - [unary](#4f1mdlm)
    - [rCurry](#2u6wntf)
    - [rcurry2](#19c6y18)
    - [rcurry3](#3tbugp1)
  + [function.combinators](#28h4qwu)
    - [always](#nmf14n)
    - [bound](#37m2jsg)
    - [comparator](#1mrcu09)
    - [complement](#46r0co2)
    - [conjoin](#2lwamvv)
    - [disjoin](#111kx3o)
    - [juxt](#3l18frh)
    - [flip](#206ipza)
    - [flip2](#4k668n3)
    - [fnull](#2zbgiuw)
    - [functionalize](#1egqt2p)
    - [mapArgs](#3ygebqi)
    - [mapArgsWith](#2dlolyb)
    - [methodize](#sqyw64)
    - [pipeline](#3cqmetx)
    - [splat](#1rvwp1q)
    - [unsplat](#4bvk7pj)
    - [unsplatl](#2r0uhxc)
  + [function.iterators](#1664s55)
    - [iterators.accumulate](#3q5sasy)
    - [iterators.accumulateWithReturn](#25b2l0r)
    - [iterators.drop](#kgcv8k)
    - [iterators.foldl](#34g0dwd)
    - [iterators.K](#1jlao46)
    - [iterators.List](#43ky6rz)
    - [iterators.map](#2iq8gzs)
    - [iterators.mapcat](#xvir7l)
    - [iterators.numbers](#3hv69ve)
    - [iterators.range](#1x0gk37)
    - [iterators.reject](#4h042r0)
    - [iterators.select](#2w5ecyt)
    - [iterators.slice](#1baon6m)
    - [iterators.take](#3vac5uf)
    - [iterators.Tree](#2afmg28)
    - [iterators.unfold](#pkwqa1)
    - [iterators.unfoldWithReturn](#39kk8xu)
  + [function.predicates](#1opuj5n)
    - [isAssociative](#48pi1tg)
    - [isDecreasing](#2nusc19)
    - [isEven](#1302m92)
    - [isFloat](#3mzq4wv)
    - [isIncreasing](#2250f4o)
    - [isIndexed](#haapch)
    - [isInstanceOf](#319y80a)
    - [isInteger](#1gf8i83)
    - [isJSON](#40ew0vw)
    - [isNegative](#2fk6b3p)
    - [isNumeric](#upglbi)
    - [isOdd](#3ep43zb)
    - [isPlainObject](#1tuee74)
    - [isPositive](#4du1wux)
    - [isSequential](#2szc72q)
    - [isValidDate](#184mhaj)
    - [isZero](#3s49zyc)
  + [object.builders](#279ka65)
    - [frequencies](#meukdy)
    - [merge](#36ei31r)
    - [renameKeys](#1ljsd9k)
    - [setPath](#45jfvxd)
    - [snapshot](#2koq656)
  + [object.selectors](#zu0gcz)
    - [accessor](#3jtnz0s)
    - [dictionary](#1yyy98l)
    - [getPath](#4iylrwe)
    - [hasPath](#2y3w247)
    - [kv](#1d96cc0)
    - [omitWhen](#3x8tuzt)
    - [pickWhen](#2ce457m)
    - [selectKeys](#rjefff)
  + [util.existential](#3bj1y38)
    - [exists](#1qoc8b1)
    - [falsey](#4anzqyu)
    - [firstExisting](#2pta16n)
    - [not](#14ykbeg)
    - [truthy](#3oy7u29)
  + [util.operators](#243i4a2)
    - [add](#j8sehv)
    - [bitwiseAnd](#338fx5o)
    - [bitwiseLeft](#1idq7dh)
    - [bitwiseRight](#42ddq1a)
    - [bitwiseNot](#2hio093)
    - [bitwiseOr](#wnyagw)
    - [bitwiseXor](#3gnlt4p)
    - [bitwiseZ](#1vsw3ci)
    - [dec](#4fsjm0b)
    - [div](#2uxtw84)
    - [eq](#1a346fx)
    - [gt](#3u2rp3q)
    - [gte](#2981zbj)
    - [inc](#odc9jc)
    - [lt](#38czs75)
    - [lte](#1nia2ey)
    - [mul](#47hxl2r)
    - [mod](#2mn7vak)
    - [neg](#11si5id)
    - [neq](#3ls5o66)
    - [not](#20xfydz)
    - [seq](#4kx3h1s)
    - [sneq](#302dr9l)
    - [sub](#1f7o1he)
  + [util.strings](#3z7bk57)
    - [camelCase](#2eclud0)
    - [explode](#thw4kt)
    - [fromQuery](#3dhjn8m)
    - [implode](#1smtxgf)
    - [strContains](#4cmhg48)
    - [toDash](#2rrrqc1)
    - [toQuery](#16x20ju)
  + [util.trampolines](#3qwpj7n)
  + [Changelog](#261ztfg)
    - [0.3.0](#l7a3n9)

# Underscore-contrib (0.3.0)

The brass buckles on Underscore's utility belt - a contributors' library for [Underscore](http://underscorejs.org/).

## [**Introduction**](#gjdgxs)

### [**Places**](#30j0zll)

* [Documentation](http://documentcloud.github.io/underscore-contrib/)
* [Source repository](https://github.com/documentcloud/underscore-contrib)
* [Tickets and bug reports](https://github.com/documentcloud/underscore-contrib/issues?state=open)
* [Maintainer's website](http://www.fogus.me)

### [**Why underscore-contrib?**](#1fob9te)

While Underscore provides a bevy of useful tools to support functional programming in JavaScript, it can't (and shouldn't) be everything to everyone. Underscore-contrib is intended as a home for functions that, for various reasons, don't belong in Underscore proper. In particular, it aims to be:

* a home for functions that are limited in scope, but solve certain point problems, and
* a proving ground for features that belong in Underscore proper, but need some advocacy and/or evolution (or devolution) to get them there.

### [**Use**](#3znysh7)

#### [**In the Browser**](#2et92p0)

First, you'll need Underscore **version 1.6.0 or higher**. Then you can grab the relevant underscore-contrib sub-libraries and simply add something like the following to your pages:

<script type="text/javascript" src="underscore.js"></script>  
<script type="text/javascript" src="underscore.object.builders.js"></script>

At the moment there are no cross-contrib dependencies (i.e. each sub-library can stand by itself), but that may change in the future.

#### [**In Node.js**](#tyjcwt)

Using contrib in Node is very simple. Just install it with npm:

npm install underscore-contrib --save

Then require it within your project like so:

var \_ = require('underscore-contrib');

The \_ variable will be a copy of Underscore with contrib's methods already mixed in.

### [**License**](#3dy6vkm)

\_.contrib is open sourced under the [MIT license](https://github.com/documentcloud/underscore-contrib/blob/master/LICENSE).

## [**Sub-libraries**](#1t3h5sf)

The \_.contrib library currently contains a number of related capabilities, aggregated into the following files.

* [underscore.array.builders](#4d34og8) - functions to build arrays
* [underscore.array.selectors](#1ci93xb) - functions to take things from arrays
* [underscore.collections.walk](#3o7alnk) - functions to walk and transform nested JavaScript objects
* [underscore.function.arity](#23ckvvd) - functions to manipulate and fix function argument arity
* [underscore.function.combinators](#28h4qwu) - functions to combine functions to make new functions
* [underscore.function.iterators](#1664s55) - functions to lazily produce, manipulate and consume sequence iterators
* [underscore.function.predicates](#1opuj5n) - functions that return true or false based on some criteria
* [underscore.object.builders](#279ka65) - functions to build JavaScript objects
* [underscore.object.selectors](#zu0gcz) - functions to pick things from JavaScript objects
* [underscore.util.existential](#3bj1y38) - functions that check for the existence or truthiness of JavaScript data types
* [underscore.util.operators](#243i4a2) - functions that wrap common (or missing) JavaScript operators
* [underscore.util.strings](#3z7bk57) - functions to work with strings
* [underscore.util.trampolines](#3qwpj7n) - functions to facilitate calling functions recursively without blowing the stack

The links above are to the annotated source code. Full-blown \_.contrib documentation is in the works. Contributors welcomed.

### [**array.builders**](#4d34og8)

Functions to build arrays. [View Annotated Source](http://docs.google.com/docs/underscore.array.builders.js.html)

#### [**cat**](#2s8eyo1)

Signature: \_.cat(... arrays:Array ...)

The \_.cat function provides a way to concatenate zero or more heterogeneous arrays into one.

\_.cat(); // 0-args  
//=> []  
  
\_.cat([]); // 1-arg, empty array  
//=> []  
  
\_.cat([1,2,3]); // 1-arg  
//=> [1,2,3]  
  
\_.cat([1,2,3],[4,5,6]); // 2-args  
//=> [1,2,3,4,5,6]  
  
\_.cat([1,2,3],[4,5,6],[7]); // 3+ args  
//=> [1,2,3,4,5,6,7]

Not every argument to \_.cat needs to be an array; other types are accepted.

Signature: \_.cat(... elems:Any ...)

\_.cat(1,[2],3,4); // mixed args  
//=> [1,2,3,4]

The \_.cat function will also work with the arguments object as if it were an array.

Signature: \_.cat(... elems:Arguments ...)

function f(){ return \_.cat(arguments, 4,5,6); }  
  
f(1,2,3);  
//=> [1,2,3,4,5,6]

#### [**chunk**](#17dp8vu)

The \_.chunk function, by default, accepts an array and a number and splits and returns a new array representing the original array split into some number of arrays of the given size:

\_.chunk([0,1,2,3], 2);  
//=> , [[0,1],[2,3]]

If the original array cannot completely fulfill the chunk scheme then the array returned will drop the undersized final chunk:

\_.chunk([0,1,2,3,4], 2);  
//=> , [[0,1],[2,3]]

You can pass an optional third argument to \_.chunk to pad out the final array chunk should it fall short. If the value given as the third argument is *not* an array then it is repeated the needed number of times:

\_.chunk([0,1,2,3,4], 3, '\_');  
//=> , [[0,1,2],[3,'\_','\_']]

If you given an array as the optional third argument then that array is used to pad in-place as needed:

\_.chunk([0,1,2,3,4], 3, ['a', 'b']);  
//=> , [[0,1,2],[3,'a','b']]

#### [**chunkAll**](#3rdcrjn)

The \_.chunkAll function is similar to \_.chunk except for the following. First, \_.chunkAll will never drop short chunks from the end:

\_.chunkAll([0,1,2,3,4], 3);  
//=> , [[0,1,2],[3]]

Also, \_.chunkAll takes an optional third argument signifying that paritions should be built from skipped regions:

\_.chunkAll(\_.range(1), 2, 4);  
//=> [[0,1],[4,5],[8,9]]

#### [**cons**](#26in1rg)

Signature: \_.cons(head:Any, tail:Array)

The \_.cons function provides a way to "construct" a new array by taking some element and putting it at the front of another array.

\_.cons(0, []);  
//=> [0]  
  
\_.cons(1, [2]);  
//=> [1,2]  
  
\_.cons([0], [1,2,3]);  
//=> [0,1,2,3]

The \_.cons function also can be used to create pairs if the second argument is not an array.

Signature: \_.cons(head:Any, tail:Any)

\_.cons(1, 2);  
//=> [1,2]  
  
\_.cons([1], 2);  
//=> [[1],2]

Finally, \_.cons will operate with the arguments object.

Signature: \_.cons(head:Any, tail:Arguments)

function f() { return \_.cons(0, arguments) }  
  
f(1,2,3);  
//=> [0,1,2,3]

#### [**cycle**](#lnxbz9)

The \_.cycle function takes an integer value used to build an array of that size containing the number of iterations through the given array, strung end-to-end as many times as needed. An example is probably more instructive:

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

#### [**interpose**](#35nkun2)

The \_.interpose function takes an array and an element and returns a new array with the given element inserted betwixt every element in the original array:

\_.interpose([1,2,3], 0);  
//=> [1,0,2,0,3]

If there are no betweens (i.e. empty and single-element arrays), then the original array is returned:

\_.interpose([1], 0);  
//=> [1]  
  
\_.interpose([], 0);  
//=> []

#### [**iterateUntil**](#1ksv4uv)

The \_.iterateUntil function takes a function used as a result generator, a function used as a stop-check and a seed value and returns an array of each generated result. The operation of \_.iterateUntil is such that the result generator is passed the seed to start and each subsequent result which will continue **until** a result fails the check function (i.e. returns falsey). An example is best:

var dec = function(n) { return n - 1; };  
var isPos = function(n) { return n > 0; };

The dec result generator takes a number and decrements it by one. The isPos predicate takes a number and returns true if it's positive. Using these two functions you can build an array of every number from 6 to 0, inclusive:

\_.iterateUntil(dec, isPos, 6);  
//=> [5,4,3,2,1]

That is, the array only contains every number from 5 down to 1 because when the result of dec got to 0 the isPos check failed (i.e. went falsey) thus terminating the execution.

#### [**keepIndexed**](#44sinio)

The \_.keepIndexed function takes an array and a function and returns a new array filled with the *non-null* return results of the given function on the elements or keys in the given array:

\_.keepIndexed([1,2,3], function(k) {   
 return i === 1 || i === 2;  
});  
  
//=> [false, true, true]

If you return either null or undefined then the result is dropped from the resulting array:

\_.keepIndexed(['a','b','c'], function(k, v) {   
 if (k === 1) return v;   
});  
  
//=> ['b']

#### [**mapcat**](#2jxsxqh)

There are times when a mapping operation produces results contained in arrays, but the final result should be flattened one level. For these circumstances you can use \_.mapcat to produce results:

var array = [1,2,null,4,undefined,6];  
  
var errors = \_.mapcat(array, function(e,i) {  
 if (e == null) {  
 return ["Element @" + i + " is bad"];  
 }  
 else {  
 return [];  
 }  
});

Inspecting the contents of errors shows:

["Element @2 is bad", "Element @4 is bad"]

The \_.mapcat function is equivalent to \_.cat.apply(array, \_.map(array,fun)).

#### [**reductions**](#z337ya)

The \_.reductions function is similar to Underscore's builtin \_.reduce function except that it returns an array of every intermediate value in the folding operation:

\_.reductions([1,2,3,4,5], function(agg, n) {  
 return agg + n;  
}, 0);  
  
//=> [1,3,6,10,15]

The last element in the array returned from \_.reductions is the answer that you would get if you had just chosen to use \_.reduce.

#### [**repeat**](#3j2qqm3)

Signature: \_.repeat(t:Integer, value:Any)

The \_.repeat function takes an integer value used to build an array of that size containing the value given:

\_.repeat(5, 'a');  
//=> ['a','a','a','a','a']

#### [**splitAt**](#1y810tw)

The \_.splitAt function takes an array and a numeric index and returns a new array with two embedded arrays representing a split of the original array at the index provided:

\_.splitAt([1,2,3,4,5], 2);  
//=> [[1,2],[3,4,5]]  
  
\_.splitAt([1,2,3,4,5], 0);  
//=> [[],[1,2,3,4,5]]

The operation of \_.splitAt is safe if the index provided is outside the range of legal indices:

\_.splitAt([1,2,3,4,5], 20000);  
//=> [[1,2,3,4,5],[]]  
  
\_.splitAt([1,2,3,4,5], -1000);  
//=> [[],[1,2,3,4,5]]   
  
\_.splitAt([], 0);  
//=> [[],[]]

#### [**takeSkipping**](#4i7ojhp)

The \_.takeSkipping function takes an array and a number and returns a new array containing every nth element in the original array:

\_.takeSkipping(\_.range(10), 2);  
//=> [0,2,4,6,8]

The \_.takeSkipping function is safe against numbers larger or smaller than the array size:

\_.takeSkipping(\_.range(10), 100000);  
//=> [0]  
  
\_.takeSkipping(\_.range(10), -100);  
//=> []

#### [**weave**](#2xcytpi)

The \_.weave function works similarly to \_.interpose (shown above) except that it accepts an array used as the interposition values. In other words, \_.weave takes two arrays and returns a new array with the original elements woven together. An example would help:

\_.weave(['a','b','c'], [1,2,3]);  
//=> ['a',1,'b',2,'c',3]

The array returned from \_.weave will be as long as the longest array given with the woven entries stopping according to the shortest array:

\_.weave(['a','b','c'], [1]);  
//=> ['a',1,'b','c']

The \_.interleave function is an alias for \_.weave.

### [**array.selectors**](#1ci93xb)

Functions to take things from arrays. [View Annotated Source](http://docs.google.com/docs/underscore.array.selectors.js.html)

#### [**best**](#3whwml4)

**Signature:** \_.best(array:Array, fun:Function)

Returns the "best" value in an array based on the result of a given function.

\_.best([1, 2, 3, 4, 5], function(x, y) {  
 return x > y;  
});  
//=> 5

#### [**dropWhile**](#2bn6wsx)

**Signature:** \_.dropWhile(array:Array, pred:Function)

Drops elements for which the given function returns a truthy value.

\_.dropWhile([-2,-1,0,1,2], isNeg);  
//=> [0,1,2]

#### [**keep**](#qsh70q)

**Signature:** \_.keep(array:Array, fun:Function)

Returns an array of existy results of a function over a source array.

\_.keep([1, 2, 3, 4, 5], function(val) {  
 if (val % 3 === 0) {  
 return val;  
 }  
});  
// => [3]

#### [**nth**](#3as4poj)

**Signature:** \_.nth(array:Array, index:Number[, guard:Any])

The \_.nth function is a convenience for the equivalent array[n]. The optional guard value allows \_.nth to work correctly as a callback for \_.map.

\_.nth(['a','b','c'], 2);  
//=> 'c'

If given an index out of bounds then \_.nth will return undefined:

\_.nth(['a','b','c'], 2000);  
//=> undefined

The \_.nth function can also be used in conjunction with \_.map and \_.compact like so:

var b = [['a'],['b'],[]];  
  
\_.compact(\_.map(b, function(e) { return \_.nth(e,0) }));  
//=> ['a','b']

If wrapping a function around \_.nth is too tedious or you'd like to partially apply the index then Underscore-contrib offers any of \_.flip2, \_.fix or rcurry2 to solve this.

#### [**partitionBy**](#1pxezwc)

**Signature:** \_.keep(array:Array, fun:Function)

Takes an array and partitions it into sub-arrays as the given predicate changes truth sense.

\_.partitionBy([1,2,2,3,1,1,5], \_.isEven);  
// => [[1],[2,2],[3,1,1,5]]  
  
\_.partitionBy([1,2,2,3,1,1,5], \_.identity);  
// => [[1],[2,2],[3],[1,1],[5]]

#### [**second**](#49x2ik5)

**Signature:** \_.second(array:Array, index:Number[, guard:Any])

The \_.second function is a convenience for the equivalent array[1]. The optional guard value allows \_.second to work correctly as a callback for \_.map.

\_.second(['a','b']);  
//=> 'b'  
  
\_.map([['a','b'], \_.range(10,20)], \_.second);  
//=> ['b',11]

You can also pass an optional number to the \_.second function to take a number of elements from an array starting with the second element and ending at the given index:

\_.second(\_.range(10), 5)  
//=> [1, 2, 3, 4]

#### [**takeWhile**](#2p2csry)

**Signature:** \_.takeWhile(array:Array, pred:Function)

The \_.takeWhile function takes an array and a function and returns a new array containing the first n elements in the original array for which the given function returns a truthy value:

var isNeg = function(n) { return n < 0; };  
  
\_.takeWhile([-2,-1,0,1,2], isNeg);  
//=> [-2,-1]

#### [**third**](#147n2zr)

**Signature:** \_.third(array:Array, index:Number[, guard:Any])

The \_.third function is a convenience for the equivalent array[2]. The optional guard value allows \_.third to work correctly as a callback for \_.map.

\_.third(['a','b','c']);  
//=> 'c'  
  
\_.map([['a','b','c'], \_.range(10,20)], \_.third);  
//=> ['c',12]

You can also pass an optional number to the \_.third function to take a number of elements from an array starting with the third element and ending at the given index:

\_.third(\_.range(10), 5)  
//=> [2, 3, 4]

### [**collections.walk**](#3o7alnk)

Functions to recursively walk collections which are trees.

Documentation should use [Journo](https://github.com/jashkenas/journo) formats and standards.

\_.walk = walk;  
 postorder: function(obj, visitor, context)  
 preorder: function(obj, visitor, context)  
 walk(obj, visitor, null, context)  
 map: function(obj, strategy, visitor, context)  
 pluck: function(obj, propertyName)  
 pluckRec: function(obj, propertyName)  
 \_.walk.collect = \_.walk.map;

### [**function.arity**](#23ckvvd)

Functions which manipulate the way functions work with their arguments.

#### [**arity**](#ihv636)

**Signature:** \_.arity(numberOfArgs:Number, fun:Function)

Returns a new function which is equivalent to fun, except that the new function's length property is equal to numberOfArgs. This does **not** limit the function to using that number of arguments. It's only effect is on the reported length.

function addAll() {  
 var sum = 0;  
 for (var i = 0; i < arguments.length; i++) {  
 sum = sum + arguments[i];  
 }  
 return sum;  
}  
  
addAll.length  
// => 0  
  
var addAllWithFixedLength = \_.arity(2, addAll);  
  
addAllWithFixedLength.length  
// => 2  
  
addAllWithFixedLength(1, 1, 1, 1);  
// => 4

#### [**binary**](#32hioqz)

**Signature:** \_.binary(fun:Function)

Returns a new function which accepts only two arguments and passes these arguments to fun. Additional arguments are discarded.

function addAll() {  
 var sum = 0;  
 for (var i = 0; i < arguments.length; i++) {  
 sum = sum + arguments[i];  
 }  
 return sum;  
}  
  
var add2 = \_.binary(addAll);  
  
add2(1, 1);  
// => 2  
  
add2(1, 1, 1, 1);  
// => 2

#### [**curry**](#1hmsyys)

**Signature:** \_.curry(func:Function[, reverse:Boolean])

Returns a curried version of func. If reverse is true, arguments will be processed from right to left.

function add3 (x, y, z) {  
 return x + y + z;  
}  
  
var curried = \_.curry(add3);  
// => function  
  
curried(1);  
// => function  
  
curried(1)(2);  
// => function  
  
curried(1)(2)(3);  
// => 6

#### [**curry2**](#41mghml)

**Signature:** \_.curry2(fun:Function)

Returns a curried version of func, but will curry exactly two arguments, no more or less.

function add2 (a, b) {  
 return a + b;  
}  
  
var curried = \_.curry2(add2);  
// => function  
  
curried(1);  
// => function  
  
curried(1)(2);  
// => 3

#### [**curry3**](#2grqrue)

**Signature:** \_.curry3(fun:Function)

Returns a curried version of func, but will curry exactly three arguments, no more or less.

function add3 (a, b, c) {  
 return a + b + c;  
}  
  
var curried = \_.curry3(add3);  
// => function  
  
curried(1);  
// => function  
  
curried(1)(2);  
// => function  
  
curried(1)(2)(3);  
// => 6

#### [**fix**](#vx1227)

**Signature:** \_.fix(fun:Function[, value:Any...])

Fixes the arguments to a function based on the parameter template defined by the presence of values and the \_ placeholder.

function add3 (a, b, c) {  
 return a + b + c;  
}  
  
var fixedFirstAndLast = \_.fix(add3, 1, \_, 3);  
// => function  
  
fixedFirstAndLast(2);  
// => 6  
  
fixedFirstAndLast(10);  
// => 14

#### [**quaternary**](#3fwokq0)

**Signature:** \_.quaternary(fun:Function)

Returns a new function which accepts only four arguments and passes these arguments to fun. Additional arguments are discarded.

function addAll() {  
 var sum = 0;  
 for (var i = 0; i < arguments.length; i++) {  
 sum = sum + arguments[i];  
 }  
 return sum;  
}  
  
var add4 = \_.quaternary(addAll);  
  
add4(1, 1, 1, 1);  
// => 4  
  
add4(1, 1, 1, 1, 1, 1);  
// => 4

#### [**ternary**](#1v1yuxt)

**Signature:** \_.ternary(fun:Function)

Returns a new function which accepts only three arguments and passes these arguments to fun. Additional arguments are discarded.

function addAll() {  
 var sum = 0;  
 for (var i = 0; i < arguments.length; i++) {  
 sum = sum + arguments[i];  
 }  
 return sum;  
}  
  
var add3 = \_.ternary(addAll);  
  
add3(1, 1, 1);  
// => 3  
  
add3(1, 1, 1, 1, 1, 1);  
// => 3

#### [**unary**](#4f1mdlm)

**Signature:** \_.unary(fun:Function)

Returns a new function which accepts only one argument and passes this argument to fun. Additional arguments are discarded.

function logArgs() {  
 console.log(arguments);  
}  
  
var logOneArg = \_.unary(logArgs);  
  
logOneArg("first");  
// => ["first"]  
  
logOneArg("first", "second");  
// => ["first"]

#### [**rCurry**](#2u6wntf)

**Signature:** \_.rCurry(func:Function)

Returns a curried version of func where arguments are processed from right to left.

function divide (a, b) {  
 return a / b;  
}  
  
var curried = \_.rCurry(divide);  
// => function  
  
curried(1);  
// => function  
  
curried(1)(2);  
// => 2  
  
curried(2)(1);  
// => 0.5

#### [**rcurry2**](#19c6y18)

**Signature:** \_.rcurry2(func:Function)

Returns a curried version of func where a maxium of two arguments are processed from right to left.

function concat () {  
 var str = "";  
  
 for (var i = 0; i < arguments.length; i++) {  
 str = str + arguments[i];  
 }  
  
 return str;  
}  
  
var curried = \_.rcurry2(concat);  
// => function  
  
curried("a");  
// => function  
  
curried("a")("b");  
// => "ba"

#### [**rcurry3**](#3tbugp1)

**Signature:** \_.rcurry3(func:Function)

Returns a curried version of func where a maxium of three arguments are processed from right to left.

function concat () {  
 var str = "";  
  
 for (var i = 0; i < arguments.length; i++) {  
 str = str + arguments[i];  
 }  
  
 return str;  
}  
  
var curried = \_.rcurry3(concat);  
// => function  
  
curried("a");  
// => function  
  
curried("a")("b");  
// => function  
  
curried("a")("b")("c");  
// => "cba"

### [**function.combinators**](#28h4qwu)

Functions that are combinators.

#### [**always**](#nmf14n)

**Signature:** \_.always(value:Any)

**Aliases:** \_.k

Takes value and returns a function that will always return value.

var platonicForm = \_.always("Eternal & Unchangeable");  
  
platonicForm();  
// => "Eternal & Unchangeable"

#### [**bound**](#37m2jsg)

**Signature:** \_.bound(obj:Object, fname:String)

Returns function property of an object by name, bound to object.

var aristotle = {  
 name: "Aristotle",  
 telos: "flourishing",  
 stateTelos: function() {  
 return this.name + "'s telos is " + this.telos;  
 }  
};  
  
var stateAristotlesTelos = \_.bound(aristotle, "stateTelos");  
  
stateAristotlesTelos();  
// => "Aristotle's Telos is flourishing"

#### [**comparator**](#1mrcu09)

**Signature:** \_.comparator(fun:Function)

Takes a binary predicate-like function and returns a comparator function (return values of -1, 0, 1) which can be used as a callback for \_.sort or Array.prototype.sort.

var lessOrEqual = function(x, y) { return x <= y; };  
  
var arr = [0, 1, -2];  
  
arr.sort(\_.comparator(lessOrEqual));  
// => [-2, 0, 1]

#### [**complement**](#46r0co2)

**Signature:** \_.complement(pred:Function)

Returns a function that reverses the sense of a given predicate-like.

function isAugustine (val) {  
 return val === "Augustine";  
}  
  
isNotAugustine = \_.complement(isAugustine);  
  
isNotAugustine("Dionysius");  
// => True

#### [**conjoin**](#2lwamvv)

**Signature:** \_.conjoin(pred:Function...)

Composes multiple predicates into a single predicate that checks all elements of an array for conformance to **all** of the original predicates.

function startsWithA (val) {  
 return val[0] === "A";  
}  
  
function endsWithE (val) {  
 return val[val.length - 1] === "e";  
}  
  
var names = ["Aristotle", "Aquinas", "Plato", "Augustine"];  
  
var startsAAndEndsE = \_.conjoin(startsWithA, endsWithE);  
  
startsAAndEndsE(names);  
// => ["Aristotle", "Augustine"]

#### [**disjoin**](#111kx3o)

**Signature:** \_.disjoin(pred:Function...)

Composes multiple predicates into a single predicate that checks all elements of an array for conformance to **any** of the original predicates.

function startsWithA (val) {  
 return val[0] === "A";  
}  
  
function endsWithE (val) {  
 return val[val.length - 1] === "e";  
}  
  
var names = ["Aristotle", "Aquinas", "Plato", "Augustine"];  
  
var startsAOrEndsE = \_.disjoin(startsWithA, endsWithE);  
  
startsAOrEndsE(names);  
// => ["Aristotle", "Aquinas", "Augustine"]

#### [**juxt**](#3l18frh)

**Signature:** \_.juxt(fun:Function...)

Returns a function whose return value is an array of the results of calling each of the original functions with the arguments.

function firstChar (val) {  
 return val[0];  
}  
  
function lastChar (val) {  
 return val[val.length - 1];  
}  
  
var firstAndLastChars = \_.juxt(firstChar, lastChar);  
  
firstAndLastChars("Etruria");  
// => ["E", "a"]

#### [**flip**](#206ipza)

**Signature:** \_.flip(fun:Function)

Returns a function that works identically to fun, but accepts the arguments in reverse order.

function regionCapitol (region, capitol) {  
 return "The capitol of " + region + " is " + capitol;  
}  
  
capitolRegion = \_.flip(regionCapitol);  
  
capitolRegion("Thessalonica", "Illyrica");  
// => "The capitol of Illyrica is Thessalonica"

#### [**flip2**](#4k668n3)

**Signature:** \_.flip2(fun:Function)

Returns a function that works identically to fun, but accepts the first two arguments in reverse order. The order of all other arguments remains the same.

function regionCapitol (region, capitol) {  
 return "The capitol of " + region + " is " + capitol;  
}  
  
capitolRegion = \_.flip2(regionCapitol);  
  
capitolRegion("Thessalonica", "Illyrica");  
// => "The capitol of Illyrica is Thessalonica"

#### [**fnull**](#2zbgiuw)

**Signature:** \_.fnull(fun:Function[, default:Any...])

Returns a function that protects fun from receiving non-existy values. Each subsequent value provided to fnull acts as the default to the original fun should a call receive non-existy values in the defaulted arg slots.

function getLength (val) {  
 return val.length;  
}  
  
safeGetLength = \_.fnull(getLength, []);  
  
safeGetLength([1, 2, 3]);  
// => 3  
  
safeGetLength(null);  
// => 0

#### [**functionalize**](#1egqt2p)

**Signature:** \_.functionalize(fun:Function[, default:Any...])

Takes a method-style function (one which uses this) and pushes this into the argument list. The returned function uses its first argument as the receiver/context of the original function, and the rest of the arguments are used as the original's entire argument list.

var militaryUnits = {  
 centuria: "80 men",  
 cohort: "480 men",  
 getDescription: function (unitName) {  
 return this[unitName];  
 }  
};  
  
var getDescription = \_.functionalize(militaryUnits.getDescription);  
  
var rulers = {  
 Leonidas: "King of Sparta",  
 Augustus: "First Roman Emperor"  
};  
  
getDescription(rulers, "Augustus");  
// => "First Roman Emperor"

#### [**mapArgs**](#3ygebqi)

**Signature:** \_.mapArgs(fun:Function)

Takes a target function and returns a new function which accepts a mapping function, which in turn returns a function that will map its arguments before calling the original target function.

function doubleNum (x) { return 2 \* x; }  
  
function squareNum (x) { return x \* x; }  
  
var squareThenDouble = \_.mapArgs(doubleNum)(squareNum);  
  
squareThenDouble(3);  
// => 18

#### [**mapArgsWith**](#2dlolyb)

**Signature:** \_.mapArgs(mapFun:Function)

Takes a mapping function and returns a new combinator function which will take a target function and return a new version which maps its arguments with the mapping function before executing the body of the target function.

function doubleNum (x) { return 2 \* x; }  
  
function squareNum (x) { return x \* x; }  
  
var squareArgs = \_.mapArgsWith(squareNum);  
  
var squareThenDouble = squareArgs(doubleNum);  
  
squareThenDouble(3);  
// => 18

#### [**methodize**](#sqyw64)

**Signature:** \_.methodize(func:Function)

Takes a function and pulls the first argument out of the argument list and into this position. The returned function calls the original with its receiver (this) prepending the argument list. The original is called with a receiver of null.

function describe (obj) {  
 return obj.name + ": " + obj.description;  
}  
  
var democritus = {  
 name: "Democritus",  
 description: "originator of the atomic hypothesis",  
 describe: \_.methodize(describe)  
};  
  
democritus.describe();  
// => "Democritus: originator of the atomic hypothesis"

#### [**pipeline**](#3cqmetx)

**Signature:** \_.pipeline(func:Function[, func2:Function...]) or \_.pipeline(funcArr:Array)

**Aliases:** \_.t

Takes a list of functions, either as an array or as individual arguments and returns a function that takes some value as its first argument and runs it through a pipeline of the original functions given.

function halveNum (x) { return x / 2; };  
function squareNum (x) { return x \* x; };  
function doubleNum (x) { return 2 \* x; };  
  
var halveSquareDouble = \_.pipeline(halveNum, squareNum, doubleNum);  
  
halveSquareDouble(1);  
// => 0.5  
  
var doubleSquareHalve = \_.pipeline([doubleNum, squareNum, halveNum]);  
  
doubleSquareHalve(1);  
// => 2

#### [**splat**](#1rvwp1q)

**Signature:** \_.splat(fun:Function)

Takes a function expecting one or more arguments and returns a function that takes an array and uses its elements as the arguments to the original function. This roughly corresponds to the [spread operator](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Spread_operator) in ECMAScript 6.

function listTwoNames (a, b) {  
 return a.name + " & " + b.name;  
}  
  
var listTwoNamesFromArray = \_.splat(listTwoNames);  
  
listTwoNamesFromArray([{ name: "Zeno" }, { name: "Parmenides"}]);  
// => "Zeno & Parmenides"

#### [**unsplat**](#4bvk7pj)

**Signature:** \_.unsplat(fun:Function)

**Aliases:** \_.unsplatr

Takes a function expecting an array as its *last* argument and returns a function which works identically, but takes a list of trailing arguments instead. Roughly corresponds to [rest parameters](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Functions_and_function_scope/rest_parameters) in ECMAScript 6.

function joinWith (joiner, arr) {  
 return arr.join(joiner);  
}  
  
var joinArgsWith = \_.unsplat(joinWith);  
  
joinArgsWith(" & ", "Plutarch", "Proclus");  
// => "Plutarch & Proclus"

#### [**unsplatl**](#2r0uhxc)

**Signature:** \_.unsplatl(fun:Function)

Similar to [unsplat](#4bvk7pj), but takes a function expecting an array as its *first* argument and returns a function which works identically, but takes a list of leading arguments instead. Roughly corresponds to [rest parameters](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Functions_and_function_scope/rest_parameters) in ECMAScript 6.

function joinWith (arr, joiner) {  
 return arr.join(joiner);  
}  
  
var joinArgsWith = \_.unsplat(joinWith);  
  
joinArgsWith("Olympiodorus", "Syrianus", " & ");  
// => "Olympiodorus & Syrianus"

### [**function.iterators**](#1664s55)

Functions to iterate over collections.

#### [**iterators.accumulate**](#3q5sasy)

**Signature:** \_.iterators.accumulate(iter:Function, binaryFn:Function, initial:Any)

Returns a function that when called will iterate one step with iter and return the value currently accumulated by using binaryFn. The function will return undefined once all values have been iterated over.

var generalsIterator = \_.iterators.List(["Hannibal", "Scipio"]);  
  
function countLetters(memo, element) {  
 return memo + element.length;  
}  
  
var generalsAcc = \_.iterators.accumulate(generalsIterator, countLetters, 0);  
  
generalsAcc();  
// => 8  
  
generalsAcc();  
// => 14  
  
generalsAcc();  
// => undefined

#### [**iterators.accumulateWithReturn**](#25b2l0r)

**Signature:** \_.iterators.accumulateWithReturn(iter:Function, binaryFn:Function, initial:Any)

Acts similarly to accumulate, except that binaryFn is expected to return an array of two elements. The value of the first element is given to the next run of binaryFn. The value of the second element is yielded by the iterator.

var fiveIter = \_.iterators.List([1, 2, 3, 4, 5]);  
  
function adderWithMessage (state, element) {  
 return [state + element, 'Total is ' + (state + element)];  
}  
  
var i = \_.iterators.accumulateWithReturn(fiveIter, adderWithMessage, 0);  
  
i();  
// => "Total is 1"  
  
i();  
// => "Total is 3"  
  
i();  
// => "Total is 6"

#### [**iterators.drop**](#kgcv8k)

**Signature:** \_.iterators.drop(iter:Function[, numberToDrop:Number])

Given an iterator function iter, will return a new iterator which iterates over the same values as iter, except that the first numberToDrop values will be omitted. If numberToDrop is not provided, it will default to 1.

var deityIter = \_.iterators.List(["Zeus", "Apollo", "Athena", "Aphrodite"]);  
  
var goddessIter = \_.iterators.drop(deityIter, 2);  
  
goddessIter();  
// => "Athena"  
  
goddessIter();  
// => "Aphrodite"

#### [**iterators.foldl**](#34g0dwd)

**Signature:** \_.iterators.foldl(iter:Function, binaryFn:Function[, seed:Any])

**Aliases:** iterators.reduce

Boils down the values given by iter into a single value. The seed is the initial state. The binaryFn is given two arguments: the seed and the current value yielded by iter.

var sybylIter = \_.iterators.List(["cumaean", "tiburtine"]);  
  
function commaString (a, b) { return a + ", " + b; }  
  
\_.iterators.foldl(sybylIter, commaString);  
// => "cumaean, tiburtine"

#### [**iterators.K**](#1jlao46)

**Signature:** \_.iterators.K(value:Any)

**Aliases:** iterators.constant

Returns a function that when invoked will always return value.

var ceasar = \_.iterators.K("Ceasar");  
  
ceasar();  
// => "ceasar"

#### [**iterators.List**](#43ky6rz)

**Signature:** \_.iterators.List(array:Array)

Returns an iterater that when invoked will iterate over the contents of array.

var triumvirIter = \_.iterators.List(["Ceasar", "Pompey", "Crassus"];  
  
triumvirIter();  
// => "Ceasar"  
  
triumvirIter();  
// => "Pompey"  
  
triumvirIter();  
// => "Crassus"

#### [**iterators.map**](#2iq8gzs)

**Signature:** \_.iterators.map(iter:Function, unaryFn:Function)

Returns a new iterator function which on each iteration will return the result of running iter's current value through unaryFn.

var notablesIter = \_.iterators.List(["Socrates", "Plato"]);  
  
function postfixAthenian (val) {  
 return val + ", Athenian";  
}  
  
var notableAtheniansIter = \_.iterators.map(notablesIter, postfixAthenian);  
  
notableAtheniansIter();  
// => "Socrates, Athenian"  
  
notableAtheniansIter();  
// => "Plato, Athenian"

#### [**iterators.mapcat**](#xvir7l)

**Signature:** \_.iterators.mapcat(iter:Function, unaryFn:Function)

Returns an iterator which is the result of flattening the contents of iter, and mapping the results with unaryFn.

function naturalSmallerThan (x) {  
 return \_.iterators.List(\_.range(0, x));  
}  
  
var treeIter = \_.iterators.Tree([1, [2]]);  
  
var smallerThanIter = \_.iterators.mapcat(treeIter, naturalSmallerThan);  
  
smallerThanIter();  
// => 0  
  
smallerThanIter();  
// => 0  
  
smallerThanIter();  
// => 1

#### [**iterators.numbers**](#3hv69ve)

**Signature:** \_.iterators.numbers([start:Number])

Returns an iterator of integers which will begin with start and increment by one for each invocation. If start is not provided it will default to 1.

var twoAndUp = \_.iterators.numbers(2);  
  
twoAndUp();  
// => 2  
  
twoAndUp();  
// => 3  
  
twoAndUp();  
// => 4

#### [**iterators.range**](#1x0gk37)

**Signature:** `\_.iterators.range([from:Number, to:Number, by:Number]);

Returns an iterator whose values consist of numbers beginning with from, ending with to, in steps of size by.

var by5 = \_.iterators.range(5, Infinity, 5);  
  
by5();  
// => 5  
  
by5();  
// => 10  
  
by5();  
// => 15

#### [**iterators.reject**](#4h042r0)

**Signature:** \_.iterators.reject(iter:Function, unaryPredicatFn:Function)

Returns an iterator consisting of the values of iter which are not flagged true by unaryPredicateFn.

var philosophers = ["Anaximander", "Socrates", "Heraclitus"];  
  
var philosophersIter = \_.iterators.List(philosophers);  
  
function isSocrates (val) {  
 return val === "Socrates";  
}  
  
var preSocraticsIter = \_.iterators.reject(philosophersIter, isSocrates);  
  
preSocraticsIter()  
// => "Anaximander"  
  
preSocraticsIter()  
// => "Heraclitus"

#### [**iterators.select**](#2w5ecyt)

**Signature:** \_.iterators.select(iter:Function, unaryPredicatFn:Function)

**Aliases:** iterators.find, iteraters.filter

Returns an iterator consisting of the values of iter which are flagged true by unaryPredicateFn.

var philosophers = ["Anaximander", "Socrates", "Heraclitus"];  
  
var philosophersIter = \_.iterators.List(philosophers);  
  
function isSocrates (val) {  
 return val === "Socrates";  
}  
  
var socraticIter = \_.iterators.select(philosophersIter, isSocrates);  
  
socraticIter()  
// => "Socrates"

#### [**iterators.slice**](#1baon6m)

**Signature:** \_.iterators.slice(iter:Function, numberToDrop:Number, numberToTake:Number)

Returns an iterator whose values consist of iter's after removing numberToDrop from the head, and a maxiumum of numberToTake of the remaining. If numberToTake is not specified, all of iter's remaining values will be used.

var emperors = ["Augustus", "Tiberius", "Caligula", "Claudius"];  
  
var emperorsIter = \_.iterators.List(emperors);  
  
var middleEmperorsIter = \_.iterators.slice(emperorsIter, 1, 2);  
  
middleEmperorsIter();  
// => "Tiberius"  
  
middleEmperorsIter();  
// => "Caligula"  
  
middleEmperorsIter();  
// => undefined

#### [**iterators.take**](#3vac5uf)

**Signature:** \_.iterators.take(iter:Function[, numberToTake:Number])

Returns an iterator consisting of the first numberToTake values yielded by iter. If numberToTake is not provided, it will default to 1.

var byzantineEmperors = ["Constantine", "Constantius", "Constans"];  
  
var byzantineEmperorsIter = \_.iterators.List(byzantineEmperors);  
  
var firstTwoEmperorsIter = \_.iterators.take(byzantineEmperorsIter, 2);  
  
firstTwoEmperorsIter();  
// => "Constantine"  
  
firstTwoEmperorsIter();  
// => "Constantius"  
  
firstTwoEmperorsIter();  
// => undefined

#### [**iterators.Tree**](#2afmg28)

**Signature:** \_.iterators.Tree(array:Array);

Returns an iterator that yields the individual values of a tree array.

var rulers = ["Augustus", ["Constantine"], ["Leo", ["Charlemagne"]]];  
  
var rulersIter = \_.iterators.Tree(rulers);  
  
rulersIter();  
// => "Augustus"  
  
rulersIter();  
// => "Constantine"  
  
rulersIter();  
// => "Leo"  
  
rulersIter();  
// => "Charlemagne"

#### [**iterators.unfold**](#pkwqa1)

**Signature:** \_.iterators.unfold(seed:Any, unaryFn:Function)

function greatify (val) {  
 return val + " the Great";  
}  
  
var greatIter = \_.iterators.unfold("Constantine", greatify);  
  
greatIter();  
// => "Constantine the Great"  
  
greatIter();  
// => "Constantine the Great the Great"  
  
greatIter();  
// => "Constantine the Great the Great the Great"

#### [**iterators.unfoldWithReturn**](#39kk8xu)

**Signature:** \_.iterators.unfold(seed:Any, unaryFn:Function)

Acts similarly to unfold, except that unaryFn is expected to return an array with two elements. The value of the first element is given to the next run of unaryFn. The value of the second element is yielded by the iterator.

var i = \_.iterators.unfoldWithReturn(1, function(n) {  
 return [n + 1, n \* n];  
});  
  
i();  
// => 1  
  
i();  
// => 4  
  
i();  
// => 9

### [**function.predicates**](#1opuj5n)

Functions which return whether the input meets a condition.

#### [**isAssociative**](#48pi1tg)

**Signature:** isAssociative(value:Any)

Returns a boolean indicating whether or not the value is an associative object. An associative object is one where its elements can be accessed via a key or index (e.g. arrays, arguments, objects).

\_.isAssociative(["Athens", "Sparta"]);  
// => true  
  
\_.isAssociative(42);  
// => false

#### [**isDecreasing**](#2nusc19)

**Signature:** \_.isDecreasing(values:Any...)

Checks whether the arguments are monotonically decreasing values (i.e. whether each argument is less than the previous argument.)

\_.isDecreasing(3, 2, 1);  
// => true  
  
\_.isDecreasing(15, 12, 2);  
// => true  
  
\_.isDecreasing(2, 3);  
// => false

#### [**isEven**](#1302m92)

**Signature:** \_.isEven(value:Any)

Checks whether the value is an even number.

\_.isEven(12);  
// => true  
  
\_.isEven(3);  
// => false  
  
\_.isEven({});  
// => false

#### [**isFloat**](#3mzq4wv)

**Signature:** \_.isFloat(value:Any)

Checks whether the value is a "float." For the purposes of this function, a float is a numeric value that is not an integer. A numeric value may be a number, a string containing a number, a Number object, etc.

**NOTE:** JavaScript itself makes no distinction between integers and floats. For the purposes of this function both 1 and 1.0 are considered integers.

\_.isFloat(1.1);  
// => true  
  
\_.isFloat(1);  
// => false  
  
\_.isFloat(1.0);  
// => false  
  
\_.isFloat("2.15");  
// => true

#### [**isIncreasing**](#2250f4o)

**Signature:** \_.isIncreasing(value:Any...)

Checks whether the arguments are monotonically increasing values (i.e. each argument is greater than the previous argument.)

\_.isIncreasing(1, 12, 15);  
// => true  
  
\_.isIncreasing(1);  
// => true  
  
\_.isIncreasing(5, 4);  
// => false

#### [**isIndexed**](#haapch)

**Signature:** \_.isIndexed(value:Any)

Checks whether the value is "indexed." An indexed value is one which accepts a numerical index to access its elements. (e.g. arrays and strings)

**NOTE:** Underscore does not support cross-browser consistent use of strings as array-like values, so be wary in IE 8 when using string objects and in IE7 and earlier when using string literals & objects.

\_.isIndexed("Socrates");  
// => true  
  
\_.isIndexed({poison: "hemlock"});  
// => false

#### [**isInstanceOf**](#319y80a)

**Signature:** \_.isInstanceOf(value:Any, constructor:Function)

Checks whether the value is an instance of the constructor.

\_.isInstanceOf(new Date(), Date);  
// => true  
  
\_.isInstanceOf("Hippocrates", RegExp);  
// => false

#### [**isInteger**](#1gf8i83)

**Signature:** \_.isInteger(value:Any)

Checks whether the value is a numeric integer. A numeric value can be a string containing a number, a Number object, etc.

\_.isInteger(18);  
// => true  
  
\_.isInteger("18");  
// => true  
  
\_.isInteger(2.5);  
// => false  
  
\_.isInteger(-1);  
// => true

#### [**isJSON**](#40ew0vw)

**Signature:** \_.isJSON(value:Any)

Checks whether the value is valid JSON. [See the spec](http://www.json.org/) for more information on what constitutes valid JSON.

**NOTE:** This function relies on JSON.parse which is not available in IE7 and earlier.

\_.isJSON('{ "name": "Crockford" }');  
// => true  
  
\_.isJSON({ name: "Crocket" });  
// => false

#### [**isNegative**](#2fk6b3p)

**Signature:** \_.isNegative(value:Any)

Checks whether the value is a negative number.

\_.isNegative(-2);  
// => true  
  
\_.isNegative(5);  
// => false

#### [**isNumeric**](#upglbi)

**Signature:** \_.isNumeric(value:Any)

A numeric is something that contains a numeric value, regardless of its type. It can be a string containing a numeric value, exponential notation, a Number object, etc.

\_.isNumeric("14");  
// => true  
  
\_.isNumeric("fourteen");  
// => false

#### [**isOdd**](#3ep43zb)

**Signature:** \_.isOdd(value:Any)

Checks whether the value is an odd number.

\_.isOdd(3);  
// => true  
  
\_.isOdd(2);  
// => false  
  
\_.isOdd({});  
// => false

#### [**isPlainObject**](#1tuee74)

**Signature:** \_.isPlainObject(value:Any);

Checks whether the value is a "plain" object created as an object literal ({}) or explicitly constructed with new Object(). Instances of other constructors are **not** plain objects.

\_.isPlainObject({});  
// => true  
  
\_.isPlainObject(new Date());  
// => false  
  
\_.isPlainObject([]);  
// => false

#### [**isPositive**](#4du1wux)

**Signature:** \_.isPositive(value:Any)

Checks whether the value is a positive number.

\_.isPositive(21);  
// => true  
  
\_.isPositive(-3);  
// => false

#### [**isSequential**](#2szc72q)

**Signature:** \_.isSequential(value:Any)

Checks whether the value is a sequential composite type (i.e. arrays and arguments).

\_.isSequential(["Herodotus", "Thucidydes");  
// => true  
  
\_.isSequential(new Date);  
// => false

#### [**isValidDate**](#184mhaj)

**Signature:** \_.isValidDate(value:Any)

Checks whether the value is a valid date. That is, the value is both an instance of Date and it represents an actual date.

Warning: This function does not verify whether the original input to Date is a real date. For instance, new Date("02/30/2014") is considered a valid date because Date coerces that into a representation of 03/02/2014. To validate strings representing a date, consider using a date/time library like [Moment.js.](http://momentjs.com/)

\_.isValidDate(new Date("January 1, 1900"));  
// => true  
  
\_.isValidDate(new Date("The Last Great Time War"));  
// => false

#### [**isZero**](#3s49zyc)

**Signature:** \_.isZero(value:Any)

Checks whether the value is 0.

\_.isZero(0);  
// => true  
  
\_.isZero("Pythagoras");  
// => false

### [**object.builders**](#279ka65)

Functions to build objects.

#### [**frequencies**](#meukdy)

**Signature:** \_.frequencies(arr:Array)

Returns an object whose property keys are the values of arr's elements. The property values are a count of how many times that value appeared in arr.

var citations = ["Plato", "Aristotle", "Plotinus", "Plato"];  
  
\_.frequencies(citations);  
// => { Plato: 2, Aristotle: 1, Plotinus: 1 }

#### [**merge**](#36ei31r)

**Signature:** \_.merge(obj1:Object[, obj:Object...])

Merges two or more objects starting with the left-most and applying the keys rightward.

\_.merge({ a: "alpha" }, { b: "beta" });  
// => { a: "alpha", b: "beta" }

#### [**renameKeys**](#1ljsd9k)

**Signature:** \_.renameKeys(obj:Object, keyMap:Object)

Takes an object (obj) and a map of keys (keyMap) and returns a new object where the keys of obj have been renamed as specified in keyMap.

\_.renameKeys({ a: 1, b: 2 }, { a: "alpha", b: "beta" });  
// => { alpha: 1, beta: 2 }

#### [**setPath**](#45jfvxd)

**Signature:** \_.setPath(obj:Object, value:Any, ks:Array, defaultValue:Any)

Sets the value of a property at any depth in obj based on the path described by the ks array. If any of the properties in the ks path don't exist, they will be created with defaultValue.

\_.setPath({}, "Plotinus", ["Platonism", "Neoplatonism"], {});  
// => { Platonism: { Neoplatonism: "Plotinus" } }

#### [**snapshot**](#2koq656)

**Signature:** \_.snapshot(obj:Object)

Snapshots/clones an object deeply.

var schools = { plato: "Academy", aristotle: "Lyceum" };  
  
\_.snapshot(schools);  
// => { plato: "Academy", aristotle: "Lyceum" }  
  
schools === \_.snapshot(schools);  
// => false

**Signature:** \_.updatePath(obj:Object, fun:Function, ks:Array, defaultValue:Any)

Updates the value at any depth in a nested object based on the path described by the ks array. The function fun is called with the current value and is expected to return a replacement value. If no keys are provided, then the object itself is presented to fun. If a property in the path is missing, then it will be created with defaultValue.

var imperialize = function (val) {  
 if (val == "Republic) return "Empire";  
 else return val;  
};  
  
\_.updatePath({ rome: "Republic" }, imperialize, ["rome"]);  
// => { rome: "Empire" }

### [**object.selectors**](#zu0gcz)

Functions to select values from an object.

#### [**accessor**](#3jtnz0s)

**Signature:** \_.accessor(field:String)

Returns a function that will attempt to look up a named field in any object that it is given.

var getName = \_.accessor('name');  
  
getName({ name: 'Seneca' });  
// => 'Seneca'

#### [**dictionary**](#1yyy98l)

**Signature:** \_.dictionary(obj:Object)

Given an object, returns a function that will attempt to look up a field that it is given.

var generals = {  
 rome: "Scipio",  
 carthage: "Hannibal"  
};  
  
var getGeneralOf = \_.dictionary(generals);  
  
\_.getGeneralOf("rome");  
// => "Scipio"

#### [**getPath**](#4iylrwe)

**Signature:** \_.getPath(obj:Object, ks:String|Array)

Gets the value at any depth in a nested object based on the path described by the keys given. Keys may be given as an array or as a dot-separated string. Returns undefined if the path cannot be reached.

var countries = {  
 greece: {  
 athens: {  
 playwright: "Sophocles"  
 }  
 }  
 }  
};  
  
\_.getPath(countries, "greece.athens.playwright");  
// => "Sophocles"  
  
\_.getPath(countries, "greece.sparta.playwright");  
// => undefined  
  
\_.getPath(countries, ["greece", "athens", "playwright"]);  
// => "Sophocles"  
  
\_.getPath(countries, ["greece", "sparta", "playwright"]);  
// => undefined

#### [**hasPath**](#2y3w247)

**Signature:** \_.hasPath(obj:Object, ks:String|Array)

Returns a boolean indicating whether there is a property at the path described by the keys given. Keys may be given as an array or as a dot-separated string.

var countries = {  
 greece: {  
 athens: {  
 playwright: "Sophocles"  
 }  
 }  
 }  
};  
  
\_.hasPath(countries, "greece.athens.playwright");  
// => true  
  
\_.hasPath(countries, "greece.sparta.playwright");  
// => false  
  
\_.hasPath(countries, ["greece", "athens", "playwright"]);  
// => true  
  
\_.hasPath(countries, ["greece", "sparta", "playwright"]);  
// => false

#### [**kv**](#1d96cc0)

**Signature:** \_.kv(obj:Object, key:String)

Returns the key/value pair for a given property in an object, undefined if not found.

var playAuthor = {  
 "Medea": "Aeschylus"  
};  
  
\_.kv(playAuthor, "Medea");  
// => ["Medea", "Aeschylus"]  
  
\_.kv(playAuthor, "Hamlet");  
// => undefined

#### [**omitWhen**](#3x8tuzt)

**Signature:** \_.omitWhen(obj, pred:Function)

Returns a copy of obj omitting any properties that the predicate (pred) function returns true for. The predicat function is invoked with each property value, like so: pred(propValue).

var playwrights = {  
 euripedes: "Greece",  
 shakespere: "England"  
};  
  
\_.omitWhen(obj, function (country) { return country == "England" });  
// => { euripedes: "Greece" }

#### [**pickWhen**](#2ce457m)

**Signature:** \_.pickWhen(obj:Object, pred:Function)

Returns a copy of obj containing only properties that the predicate (pred) function returns true for. The predicate function is invoked with each property value, like so: pred(propValue).

var playwrights = {  
 euripedes: "Greece",  
 shakespere: "England"  
};  
  
\_.pickWhen(obj, function (country) { return country == "England" });  
// => { shakespeare: "England" }

#### [**selectKeys**](#rjefff)

**Signature:** `\_.selectKeys(obj:Object, ks:Array);

Returns a copy of obj containing only the properties listed in the ks array.

var philosopherCities = {  
 Philo: "Alexandria",  
 Plato: "Athens",  
 Plotinus: "Rome"  
}  
  
\_.selectKeys(philosopherCities, ["Plato", "Plotinus"]);  
// => { Plato: "Athens", Plotinus: "Rome" }

### [**util.existential**](#3bj1y38)

Functions which deal with whether a value "exists."

#### [**exists**](#1qoc8b1)

**Signature:** \_.exists(value:Any)

Checks whether or not the value is "existy." Both null and undefined are considered non-existy values. All other values are existy.

\_.exists(null);  
// => false  
  
\_.exists(undefined);  
// => false  
  
\_.exists({});  
// = > true  
  
\_.exists("Sparta");  
// => true

#### [**falsey**](#4anzqyu)

**Signature:** \_.falsey(value:Any)

Checks whether the value is falsey. A falsey value is one which coerces to false in a boolean context.

\_.falsey(0);  
// => true  
  
\_.falsey("");  
// => true  
  
\_.falsey({});  
// => false  
  
\_.falsey("Corinth");  
// => false

#### [**firstExisting**](#2pta16n)

**Signature:** \_.firstExisting(value:Any[, value:Any...])

Returns the first existy argument from the argument list.

\_.firstExisting("Socrates", "Plato");  
// => "Socrates"  
  
\_.firstExisting(null, undefined, "Heraclitus");  
// => "Heraclitus"

#### [**not**](#14ykbeg)

**Signature:** \_.not(value:Any)

Returns a boolean which is the opposite of the truthiness of the original value.

\_.not(0);  
// => true  
  
\_.not(1);  
// => false  
  
\_.not(true);  
// => false  
  
\_.not(false);  
// => true  
  
\_.not({});  
// => false  
  
\_.not(null);  
// => true

#### [**truthy**](#3oy7u29)

**Signature:** \_.truthy(value:Any)

Checks whether the value is truthy. A truthy value is one which coerces to true in a boolean context.

\_.truthy({});  
// => true  
  
\_.truthy("Athens");  
// => true  
  
\_.truthy(0);  
// => false  
  
\_.truthy("");  
// => false

### [**util.operators**](#243i4a2)

Functions which wrap JavaScript's operators.

#### [**add**](#j8sehv)

**Signature:** \_.add(value:Number, value:Number[, value:Number...])

Returns the sum of the arguments.

\_.add(1, 2, 3, 4);  
// => 10

#### [**bitwiseAnd**](#338fx5o)

**Signature:** \_.bitwiseAnd(value:Any, value:Any[, value:Any...])

Returns the result of using the & operator on the arguments.

\_.bitwiseAnd(1, 3);  
// => 1  
  
\_.bitwiseAnd(1, 3, 2);  
// => 0

#### [**bitwiseLeft**](#1idq7dh)

**Signature:** \_.bitwiseLeft(value:Any, value:Any[, value:Any...])

Returns the result of using the << operator on the arguments.

\_.bitwiseLeft(1, 3);  
// => 8  
  
\_.bitwiseLeft(1, 3, 2);  
// => 32

#### [**bitwiseRight**](#42ddq1a)

**Signature:** \_.bitwiseRight(value:Any, value:Any[, value:Any...])

Returns the result of using the >> operator on the arguments.

\_.bitwiseRight(3, 1);  
// => 1  
  
\_.bitwiseRight(3, 1, 3);  
// => 0

#### [**bitwiseNot**](#2hio093)

**Signature:** \_.bitwiseNot(value:Any)

Returns the result of using the ~ operator on the value.

\_.bitwiseNot(1);  
// => -2  
  
\_.bitwiseOr(2);  
// => -3

#### [**bitwiseOr**](#wnyagw)

**Signature:** \_.bitwiseOr(value:Any, value:Any[, value:Any...])

Returns the result of using the | operator on the arguments.

\_.bitwiseOr(1, 3);  
// => 3  
  
\_.bitwiseOr(1, 3, 4);  
// => 7

#### [**bitwiseXor**](#3gnlt4p)

**Signature:** \_.bitwiseXor(value:Any, value:Any[, value:Any...])

Returns the result of using the ^ operator on the arguments.

\_.bitwiseXor(1, 3);  
// => 2  
  
\_.bitwiseXor(1, 3, 3);  
// => 1

#### [**bitwiseZ**](#1vsw3ci)

**Signature:** \_.bitwiseZ(value:Any, value:Any[, value:Any...])

Returns the result of using the >>> operator on the arguments.

\_.bitwiseZ(72, 32);  
// => 72  
  
\_.bitwiseZ(72, 32, 2);  
// => 18

#### [**dec**](#4fsjm0b)

**Signature:** \_.dec(value:Number)

Returns the result of decrementing the value by 1.

\_.dec(2);  
// => 1

#### [**div**](#2uxtw84)

**Signature:** \_.div(value:Number, value:Number[, value:Number...])

Returns the quotient of the arguments.

\_.div(8, 2);  
// => 4  
  
\_.div(8, 2, 2);  
// => 2

#### [**eq**](#1a346fx)

**Signature:** \_.eq(value:Any, value:Any[, value:Any...])

Compares the arguments with loose equality (==).

\_.eq(1, "1");  
// => true  
  
\_.eq(1, 15);  
// => false  
  
\_.eq(1, true, "1");  
// => true  
  
\_.eq(1, 1, 15);  
// => false

#### [**gt**](#3u2rp3q)

**Signature:** \_.gt(value:Any, value:Any[, value:Any...])

Checks whether each argument is greater than the previous argument.

\_.gt(1, 2);  
// => true  
  
\_.gt(1, 2, 3);  
// => true  
  
\_.gt(1, 6, 2);  
// => false

#### [**gte**](#2981zbj)

**Signature:** \_.gte(value:Any, value:Any[, value:Any...])

Checks whether each argument is greater than or equal to the previous argument.

\_.gte(1, 2);  
// => true  
  
\_.gte(1, 1, 3);  
// => true  
  
\_.gte(1, 6, 2);  
// => false

#### [**inc**](#odc9jc)

**Signature:** \_.inc(value:Number)

Returns the result of incrementing the value by 1.

\_.inc(2);  
// => 3

#### [**lt**](#38czs75)

**Signature:** \_.lt(value:Any, value:Any[, value:Any...])

Checks whether each argument is less than the previous argument.

\_.lt(2, 1);  
// => true  
  
\_.lt(2, 1, 0);  
// => true  
  
\_.lt(2, 1, 12);  
// => false

#### [**lte**](#1nia2ey)

**Signature:** \_.lte(value:Any, value:Any[, value:Any...])

Checks whether each argument is less than or equal to the previous argument.

\_.lte(2, 1);  
// => true  
  
\_.lte(2, 1, 1);  
// => true  
  
\_.lte(2, 1, 12);  
// => false

#### [**mul**](#47hxl2r)

**Signature:** \_.mul(value:Number, value:Number[, value:Number...])

Returns the product of the arguments.

\_.mul(1, 2, 3, 4);  
// => 24

#### [**mod**](#2mn7vak)

**Signature:** \_.mod(dividend:Number, divisor:Number)

Returns the remainder of dividing dividend by divisor.

\_.mod(26, 5);  
// => 1  
  
\_.mod(14, 3);  
// => 2

#### [**neg**](#11si5id)

**Signature:** \_.neg(num:Number)

Returns a new number with the opposite sign value of num.

\_.neg(5);  
// => -5  
  
\_.neg(-3);  
// => 3

#### [**neq**](#3ls5o66)

**Signature:** \_.neq(value:Any, value:Any[, value:Any...])

Checks whether each argument is not equal to the previous argument, using loose inequality (!=).

\_.neq(2, 1);  
// => true  
  
\_.neq(2, 1, 1);  
// => true  
  
\_.neq(1, 1);  
// => false

#### [**not**](#20xfydz)

**Signature:** \_.not(value:Any)

Returns a boolean which is the opposite of the truthiness of the original value.

\_.not(0);  
// => true  
  
\_.not(1);  
// => false  
  
\_.not(true);  
// => false  
  
\_.not(false);  
// => true  
  
\_.not({});  
// => false  
  
\_.not(null);  
// => true

#### [**seq**](#4kx3h1s)

**Signature:** \_.seq(value:Any, value:Any[, value:Any...])

Checks whether the arguments are strictly equal (===) to each other.

\_.seq(2, 2);  
// => true  
  
\_.seq(2, "2");  
// => false  
  
\_.seq(2, 2, 2);  
// => true

#### [**sneq**](#302dr9l)

**Signature:** \_.sneq(value:Any, value:Any[, value:Any...])

Checks whether the arguments are strictly not equal (!==) to each other.

\_.sneq(2, 2);  
// => false  
  
\_.sneq(2, "2");  
// => true  
  
\_.sneq(2, 2, 2);  
// => false

#### [**sub**](#1f7o1he)

**Signature:** \_.sub(value:Number, value:Number[, value:Number...])

Returns the difference of the arguments.

\_.sub(10, 3);  
// => 7  
  
\_.sub(10, 3, 5);  
// => 2

### [**util.strings**](#3z7bk57)

Functions for working with strings.

#### [**camelCase**](#2eclud0)

**Signature:** \_.camelCase(string:String)

Converts a dash-separated string to camel case. Opposite of [toDash](#2rrrqc1).

\_.camelCase("ancient-greece");  
// => "ancientGreece"

#### [**explode**](#thw4kt)

**Signature:** \_.explode(s:String)

Explodes a string into an array of characters. Opposite of [implode](#1smtxgf).

\_.explode("Plato");  
// => ["P", "l", "a", "t", "o"]

#### [**fromQuery**](#3dhjn8m)

**Signature:** \_.fromQuery(str:String)

Takes a URL query string and converts it into an equivalent JavaScript object. Opposite of [toQuery](#16x20ju)

\_.fromQuery("forms%5Bperfect%5D=circle&forms%5Bimperfect%5D=square");  
// => { forms: { perfect: "circle", imperfect: "square" } }

#### [**implode**](#1smtxgf)

**Signature:** \_.implode(a:Array)

Implodes an array of strings into a single string. Opposite of [explode](#thw4kt).

\_.implode(["H", "o", "m", "e", "r"]);  
// => "Homer"

#### [**strContains**](#4cmhg48)

**Signature:** \_.strContains(str:String, search:String)

Reports whether a string contains a search string.

\_.strContains("Acropolis", "polis");  
// => true

#### [**toDash**](#2rrrqc1)

**Signature:** \_.toDash(string:String)

Converts a camel case string to a dashed string. Opposite of [camelCase](#2eclud0).

\_.toDash("thisIsSparta");  
// => "this-is-sparta"

#### [**toQuery**](#16x20ju)

**Signature:** \_.toQuery(obj:Object)

Takes an object and converts it into an equivalent URL query string. Opposite of [fromQuery](#3dhjn8m).

\_.toQuery({ forms: { perfect: "circle", imperfect: "square" } });  
// => "forms%5Bperfect%5D=circle&forms%5Bimperfect%5D=square"

### [**util.trampolines**](#3qwpj7n)

Trampoline functions.

Documentation should use [Journo](https://github.com/jashkenas/journo) formats and standards.

done: function(value) {  
trampoline: function(fun /\*, args \*/) {

### [**Changelog**](#261ztfg)

#### [**0.3.0**](#l7a3n9)

* Contrib now requires Underscore 1.6.0 or higher.
* Rename partition and partitionAll to chunk and chunkAll to avoid name conflicts with Underscore's partition - [#115](https://github.com/documentcloud/underscore-contrib/issues/115)
* Added toQuery and fromQuery - [#134](https://github.com/documentcloud/underscore-contrib/issues/134)
* Switch always to an alias of Underscore's constant. - [#132](https://github.com/documentcloud/underscore-contrib/issues/132)
* The combinators sub-library now supports method combinators - [#14](https://github.com/documentcloud/underscore-contrib/issues/14)