

[Underscore](http://github.com/jashkenas/underscore/) is a JavaScript library that provides a whole mess of useful functional programming helpers without extending any built-in objects. It’s the answer to the question: “If I sit down in front of a blank HTML page, and want to start being productive immediately, what do I need?” … and the tie to go along with [jQuery](http://jquery.com/)'s tux and [Backbone](http://backbonejs.org/)'s suspenders.

Underscore provides over 100 functions that support both your favorite workaday functional helpers: **map**, **filter**, **invoke** — as well as more specialized goodies: function binding, javascript templating, creating quick indexes, deep equality testing, and so on.

A complete [Test Suite](http://underscorejs.org/test/) is included for your perusal.

You may also read through the [annotated source code](http://underscorejs.org/docs/underscore.html).

Enjoying Underscore, and want to *turn it up to 11?* Try [Underscore-contrib](http://documentcloud.github.io/underscore-contrib/).

The project is [hosted on GitHub](https://github.com/jashkenas/underscore). You can report bugs and discuss features on the[issues page](https://github.com/jashkenas/underscore/issues), on Freenode in the #documentcloud channel, or in our [Gitter](https://gitter.im/jashkenas/underscore) channel.

*Underscore is an open-source component of*[*DocumentCloud*](http://documentcloud.org/)*.*

**Downloads *(Right-click, and use "Save As")***

|  |  |
| --- | --- |
| [Development Version (1.8.3)](http://underscorejs.org/underscore.js) | *52kb, Uncompressed with Plentiful Comments* |
| [Production Version (1.8.3)](http://underscorejs.org/underscore-min.js) | *5.7kb, Minified and Gzipped*  ([Source Map](http://underscorejs.org/underscore-min.map)) |
|  | |
| [Edge Version](https://raw.github.com/jashkenas/underscore/master/underscore.js) | *Unreleased, current*master*, use at your own risk* |

**Installation**

* **Node.js** npm install underscore
* **Meteor.js** meteor add underscore
* **Require.js** require(["underscore"], ...
* **Bower** bower install underscore
* **Component** component install jashkenas/underscore

**Collection Functions (Arrays or Objects)**

**each**\_.each(list, iteratee, [context]) *Alias:****forEach***   
Iterates over a **list** of elements, yielding each in turn to an **iteratee** function. The **iteratee** is bound to the **context** object, if one is passed. Each invocation of **iteratee** is called with three arguments: (element, index, list). If **list** is a JavaScript object, **iteratee**'s arguments will be (value, key, list). Returns the **list** for chaining.

\_.each([1, 2, 3], alert);

=> alerts each number in turn...

\_.each({one: 1, two: 2, three: 3}, alert);

=> alerts each number value in turn...

*Note: Collection functions work on arrays, objects, and array-like objects such as*arguments, NodeList*and similar. But it works by duck-typing, so avoid passing objects with a numeric*length*property. It's also good to note that an*each*loop cannot be broken out of — to break, use****\_.find****instead.*

**map**\_.map(list, iteratee, [context]) *Alias:****collect***   
Produces a new array of values by mapping each value in **list** through a transformation function (**iteratee**). The iteratee is passed three arguments: the value, then the index (or key) of the iteration, and finally a reference to the entirelist.

\_.map([1, 2, 3], function(num){ return num \* 3; });

=> [3, 6, 9]

\_.map({one: 1, two: 2, three: 3}, function(num, key){ return num \* 3; });

=> [3, 6, 9]

\_.map([[1, 2], [3, 4]], \_.first);

=> [1, 3]

**reduce**\_.reduce(list, iteratee, [memo], [context]) *Aliases:****inject****,****foldl***   
Also known as **inject** and **foldl**, reduce boils down a **list** of values into a single value.**Memo** is the initial state of the reduction, and each successive step of it should be returned by **iteratee**. The iteratee is passed four arguments: the memo, then the valueand index (or key) of the iteration, and finally a reference to the entire list.

If no memo is passed to the initial invocation of reduce, the iteratee is not invoked on the first element of the list. The first element is instead passed as the memo in the invocation of the iteratee on the next element in the list.

var sum = \_.reduce([1, 2, 3], function(memo, num){ return memo + num; }, 0);

=> 6

**reduceRight**\_.reduceRight(list, iteratee, [memo], [context]) *Alias:****foldr***   
The right-associative version of **reduce**. **Foldr** is not as useful in JavaScript as it would be in a language with lazy evaluation.

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

var flat = \_.reduceRight(list, function(a, b) { return a.concat(b); }, []);

=> [4, 5, 2, 3, 0, 1]

**find**\_.find(list, predicate, [context]) *Alias:****detect***   
Looks through each value in the **list**, returning the first one that passes a truth test (**predicate**), or undefined if no value passes the test. The function returns as soon as it finds an acceptable element, and doesn't traverse the entire list.

var even = \_.find([1, 2, 3, 4, 5, 6], function(num){ return num % 2 == 0; });

=> 2

**filter**\_.filter(list, predicate, [context]) *Alias:****select***   
Looks through each value in the **list**, returning an array of all the values that pass a truth test (**predicate**).

var evens = \_.filter([1, 2, 3, 4, 5, 6], function(num){ return num % 2 == 0; });

=> [2, 4, 6]

**where**\_.where(list, properties)   
Looks through each value in the **list**, returning an array of all the values that contain all of the key-value pairs listed in **properties**.

\_.where(listOfPlays, {author: "Shakespeare", year: 1611});

=> [{title: "Cymbeline", author: "Shakespeare", year: 1611},

{title: "The Tempest", author: "Shakespeare", year: 1611}]

**findWhere**\_.findWhere(list, properties)   
Looks through the **list** and returns the *first* value that matches all of the key-value pairs listed in **properties**.

If no match is found, or if **list** is empty, *undefined* will be returned.

\_.findWhere(publicServicePulitzers, {newsroom: "The New York Times"});

=> {year: 1918, newsroom: "The New York Times",

reason: "For its public service in publishing in full so many official reports,

documents and speeches by European statesmen relating to the progress and

conduct of the war."}

**reject**\_.reject(list, predicate, [context])   
Returns the values in **list** without the elements that the truth test (**predicate**) passes. The opposite of **filter**.

var odds = \_.reject([1, 2, 3, 4, 5, 6], function(num){ return num % 2 == 0; });

=> [1, 3, 5]

**every**\_.every(list, [predicate], [context]) *Alias:****all***   
Returns *true* if all of the values in the **list** pass the **predicate** truth test. Short-circuits and stops traversing the list if a false element is found.

\_.every([2, 4, 5], function(num) { return num % 2 == 0; });

=> false

**some**\_.some(list, [predicate], [context]) *Alias:****any***   
Returns *true* if any of the values in the **list** pass the **predicate** truth test. Short-circuits and stops traversing the list if a true element is found.

\_.some([null, 0, 'yes', false]);

=> true

**contains**\_.contains(list, value, [fromIndex]) *Alias:****includes***   
Returns *true* if the **value** is present in the **list**. Uses **indexOf** internally, if **list** is an Array. Use **fromIndex** to start your search at a given index.

\_.contains([1, 2, 3], 3);

=> true

**invoke**\_.invoke(list, methodName, \*arguments)   
Calls the method named by **methodName** on each value in the **list**. Any extra arguments passed to **invoke** will be forwarded on to the method invocation.

\_.invoke([[5, 1, 7], [3, 2, 1]], 'sort');

=> [[1, 5, 7], [1, 2, 3]]

**pluck**\_.pluck(list, propertyName)   
A convenient version of what is perhaps the most common use-case for **map**: extracting a list of property values.

var stooges = [{name: 'moe', age: 40}, {name: 'larry', age: 50}, {name: 'curly', age: 60}];

\_.pluck(stooges, 'name');

=> ["moe", "larry", "curly"]

**max**\_.max(list, [iteratee], [context])   
Returns the maximum value in **list**. If an **iteratee** function is provided, it will be used on each value to generate the criterion by which the value is ranked. *-Infinity* is returned if **list** is empty, so an [isEmpty](http://underscorejs.org/#isEmpty) guard may be required. Non-numerical values in **list** will be ignored.

var stooges = [{name: 'moe', age: 40}, {name: 'larry', age: 50}, {name: 'curly', age: 60}];

\_.max(stooges, function(stooge){ return stooge.age; });

=> {name: 'curly', age: 60};

**min**\_.min(list, [iteratee], [context])   
Returns the minimum value in **list**. If an **iteratee** function is provided, it will be used on each value to generate the criterion by which the value is ranked. *Infinity* is returned if **list** is empty, so an [isEmpty](http://underscorejs.org/#isEmpty) guard may be required. Non-numerical values in **list** will be ignored.

var numbers = [10, 5, 100, 2, 1000];

\_.min(numbers);

=> 2

**sortBy**\_.sortBy(list, iteratee, [context])   
Returns a (stably) sorted copy of **list**, ranked in ascending order by the results of running each value through **iteratee**. iteratee may also be the string name of the property to sort by (eg. length).

\_.sortBy([1, 2, 3, 4, 5, 6], function(num){ return Math.sin(num); });

=> [5, 4, 6, 3, 1, 2]

var stooges = [{name: 'moe', age: 40}, {name: 'larry', age: 50}, {name: 'curly', age: 60}];

\_.sortBy(stooges, 'name');

=> [{name: 'curly', age: 60}, {name: 'larry', age: 50}, {name: 'moe', age: 40}];

**groupBy**\_.groupBy(list, iteratee, [context])   
Splits a collection into sets, grouped by the result of running each value through **iteratee**. If **iteratee** is a string instead of a function, groups by the property named by **iteratee** on each of the values.

\_.groupBy([1.3, 2.1, 2.4], function(num){ return Math.floor(num); });

=> {1: [1.3], 2: [2.1, 2.4]}

\_.groupBy(['one', 'two', 'three'], 'length');

=> {3: ["one", "two"], 5: ["three"]}

**indexBy**\_.indexBy(list, iteratee, [context])   
Given a **list**, and an **iteratee** function that returns a key for each element in the list (or a property name), returns an object with an index of each item. Just like [groupBy](http://underscorejs.org/#groupBy), but for when you know your keys are unique.

var stooges = [{name: 'moe', age: 40}, {name: 'larry', age: 50}, {name: 'curly', age: 60}];

\_.indexBy(stooges, 'age');

=> {

"40": {name: 'moe', age: 40},

"50": {name: 'larry', age: 50},

"60": {name: 'curly', age: 60}

}

**countBy**\_.countBy(list, iteratee, [context])   
Sorts a list into groups and returns a count for the number of objects in each group. Similar to groupBy, but instead of returning a list of values, returns a count for the number of values in that group.

\_.countBy([1, 2, 3, 4, 5], function(num) {

return num % 2 == 0 ? 'even': 'odd';

});

=> {odd: 3, even: 2}

**shuffle**\_.shuffle(list)   
Returns a shuffled copy of the **list**, using a version of the [Fisher-Yates shuffle](http://en.wikipedia.org/wiki/Fisher%E2%80%93Yates_shuffle).

\_.shuffle([1, 2, 3, 4, 5, 6]);

=> [4, 1, 6, 3, 5, 2]

**sample**\_.sample(list, [n])   
Produce a random sample from the **list**. Pass a number to return **n** random elements from the list. Otherwise a single random item will be returned.

\_.sample([1, 2, 3, 4, 5, 6]);

=> 4

\_.sample([1, 2, 3, 4, 5, 6], 3);

=> [1, 6, 2]

**toArray**\_.toArray(list)   
Creates a real Array from the **list** (anything that can be iterated over). Useful for transmuting the **arguments** object.

(function(){ return \_.toArray(arguments).slice(1); })(1, 2, 3, 4);

=> [2, 3, 4]

**size**\_.size(list)   
Return the number of values in the **list**.

\_.size({one: 1, two: 2, three: 3});

=> 3

**partition**\_.partition(array, predicate)   
Split **array** into two arrays: one whose elements all satisfy **predicate** and one whose elements all do not satisfy **predicate**.

\_.partition([0, 1, 2, 3, 4, 5], isOdd);

=> [[1, 3, 5], [0, 2, 4]]

**Array Functions**

*Note: All array functions will also work on the****arguments****object. However, Underscore functions are not designed to work on "sparse" arrays.*

**first**\_.first(array, [n]) *Aliases:****head****,****take***   
Returns the first element of an **array**. Passing **n** will return the first **n** elements of the array.

\_.first([5, 4, 3, 2, 1]);

=> 5

**initial**\_.initial(array, [n])   
Returns everything but the last entry of the array. Especially useful on the arguments object. Pass **n** to exclude the last **n** elements from the result.

\_.initial([5, 4, 3, 2, 1]);

=> [5, 4, 3, 2]

**last**\_.last(array, [n])   
Returns the last element of an **array**. Passing **n** will return the last **n** elements of the array.

\_.last([5, 4, 3, 2, 1]);

=> 1

**rest**\_.rest(array, [index]) *Aliases:****tail****,****drop***   
Returns the **rest** of the elements in an array. Pass an **index** to return the values of the array from that index onward.

\_.rest([5, 4, 3, 2, 1]);

=> [4, 3, 2, 1]

**compact**\_.compact(array)   
Returns a copy of the **array** with all falsy values removed. In JavaScript, *false*, *null*, *0*, *""*, *undefined* and *NaN* are all falsy.

\_.compact([0, 1, false, 2, '', 3]);

=> [1, 2, 3]

**flatten**\_.flatten(array, [shallow])   
Flattens a nested **array** (the nesting can be to any depth). If you pass **shallow**, the array will only be flattened a single level.

\_.flatten([1, [2], [3, [[4]]]]);

=> [1, 2, 3, 4];

\_.flatten([1, [2], [3, [[4]]]], true);

=> [1, 2, 3, [[4]]];

**without**\_.without(array, \*values)   
Returns a copy of the **array** with all instances of the **values** removed.

\_.without([1, 2, 1, 0, 3, 1, 4], 0, 1);

=> [2, 3, 4]

**union**\_.union(\*arrays)   
Computes the union of the passed-in **arrays**: the list of unique items, in order, that are present in one or more of the **arrays**.

\_.union([1, 2, 3], [101, 2, 1, 10], [2, 1]);

=> [1, 2, 3, 101, 10]

**intersection**\_.intersection(\*arrays)   
Computes the list of values that are the intersection of all the **arrays**. Each value in the result is present in each of the **arrays**.

\_.intersection([1, 2, 3], [101, 2, 1, 10], [2, 1]);

=> [1, 2]

**difference**\_.difference(array, \*others)   
Similar to **without**, but returns the values from **array** that are not present in the **other**arrays.

\_.difference([1, 2, 3, 4, 5], [5, 2, 10]);

=> [1, 3, 4]

**uniq**\_.uniq(array, [isSorted], [iteratee]) *Alias:****unique***   
Produces a duplicate-free version of the **array**, using *===* to test object equality. In particular only the first occurence of each value is kept. If you know in advance that the **array** is sorted, passing *true* for **isSorted** will run a much faster algorithm. If you want to compute unique items based on a transformation, pass an **iteratee** function.

\_.uniq([1, 2, 1, 4, 1, 3]);

=> [1, 2, 4, 3]

**zip**\_.zip(\*arrays)   
Merges together the values of each of the **arrays** with the values at the corresponding position. Useful when you have separate data sources that are coordinated through matching array indexes. Use with apply to pass in an array of arrays. If you're working with a matrix of nested arrays, this can be used to transpose the matrix.

\_.zip(['moe', 'larry', 'curly'], [30, 40, 50], [true, false, false]);

=> [["moe", 30, true], ["larry", 40, false], ["curly", 50, false]]

**unzip**\_.unzip(array)   
The opposite of [zip](http://underscorejs.org/#zip). Given an **array** of arrays, returns a series of new arrays, the first of which contains all of the first elements in the input arrays, the second of which contains all of the second elements, and so on.

\_.unzip([["moe", 30, true], ["larry", 40, false], ["curly", 50, false]]);

=> [['moe', 'larry', 'curly'], [30, 40, 50], [true, false, false]]

**object**\_.object(list, [values])   
Converts arrays into objects. Pass either a single list of [key, value] pairs, or a list of keys, and a list of values. If duplicate keys exist, the last value wins.

\_.object(['moe', 'larry', 'curly'], [30, 40, 50]);

=> {moe: 30, larry: 40, curly: 50}

\_.object([['moe', 30], ['larry', 40], ['curly', 50]]);

=> {moe: 30, larry: 40, curly: 50}

**indexOf**\_.indexOf(array, value, [isSorted])   
Returns the index at which **value** can be found in the **array**, or *-1* if value is not present in the **array**. If you're working with a large array, and you know that the array is already sorted, pass true for **isSorted** to use a faster binary search ... or, pass a number as the third argument in order to look for the first matching value in the array after the given index.

\_.indexOf([1, 2, 3], 2);

=> 1

**lastIndexOf**\_.lastIndexOf(array, value, [fromIndex])   
Returns the index of the last occurrence of **value** in the **array**, or *-1* if value is not present. Pass **fromIndex** to start your search at a given index.

\_.lastIndexOf([1, 2, 3, 1, 2, 3], 2);

=> 4

**sortedIndex**\_.sortedIndex(list, value, [iteratee], [context])   
Uses a binary search to determine the index at which the **value** *should* be inserted into the **list** in order to maintain the **list**'s sorted order. If an **iteratee** function is provided, it will be used to compute the sort ranking of each value, including the **value** you pass. The iteratee may also be the string name of the property to sort by (eg. length).

\_.sortedIndex([10, 20, 30, 40, 50], 35);

=> 3

var stooges = [{name: 'moe', age: 40}, {name: 'curly', age: 60}];

\_.sortedIndex(stooges, {name: 'larry', age: 50}, 'age');

=> 1

**findIndex**\_.findIndex(array, predicate, [context])   
Similar to [\_.indexOf](http://underscorejs.org/#indexOf), returns the first index where the **predicate** truth test passes; otherwise returns *-1*.

\_.findIndex([4, 6, 8, 12], isPrime);

=> -1 // not found

\_.findIndex([4, 6, 7, 12], isPrime);

=> 2

**findLastIndex**\_.findLastIndex(array, predicate, [context])   
Like [\_.findIndex](http://underscorejs.org/#findIndex) but iterates the array in reverse, returning the index closest to the end where the **predicate** truth test passes.

var users = [{'id': 1, 'name': 'Bob', 'last': 'Brown'},

{'id': 2, 'name': 'Ted', 'last': 'White'},

{'id': 3, 'name': 'Frank', 'last': 'James'},

{'id': 4, 'name': 'Ted', 'last': 'Jones'}];

\_.findLastIndex(users, {

name: 'Ted'

});

=> 3

**range**\_.range([start], stop, [step])   
A function to create flexibly-numbered lists of integers, handy for each and map loops. **start**, if omitted, defaults to *0*; **step** defaults to *1*. Returns a list of integers from **start** (inclusive) to **stop** (exclusive), incremented (or decremented) by **step**, exclusive. Note that ranges that **stop** before they **start** are considered to be zero-length instead of negative — if you'd like a negative range, use a negative **step**.

\_.range(10);

=> [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

\_.range(1, 11);

=> [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

\_.range(0, 30, 5);

=> [0, 5, 10, 15, 20, 25]

\_.range(0, -10, -1);

=> [0, -1, -2, -3, -4, -5, -6, -7, -8, -9]

\_.range(0);

=> []

**Function (uh, ahem) Functions**

**bind**\_.bind(function, object, \*arguments)   
Bind a **function** to an **object**, meaning that whenever the function is called, the value of *this* will be the **object**. Optionally, pass **arguments** to the **function** to pre-fill them, also known as **partial application**. For partial application without context binding, use [partial](http://underscorejs.org/#partial).

var func = function(greeting){ return greeting + ': ' + this.name };

func = \_.bind(func, {name: 'moe'}, 'hi');

func();

=> 'hi: moe'

**bindAll**\_.bindAll(object, \*methodNames)   
Binds a number of methods on the **object**, specified by **methodNames**, to be run in the context of that object whenever they are invoked. Very handy for binding functions that are going to be used as event handlers, which would otherwise be invoked with a fairly useless *this*. **methodNames** are required.

var buttonView = {

label : 'underscore',

onClick: function(){ alert('clicked: ' + this.label); },

onHover: function(){ console.log('hovering: ' + this.label); }

};

\_.bindAll(buttonView, 'onClick', 'onHover');

// When the button is clicked, this.label will have the correct value.

jQuery('#underscore\_button').on('click', buttonView.onClick);

**partial**\_.partial(function, \*arguments)   
Partially apply a function by filling in any number of its **arguments**, *without* changing its dynamic this value. A close cousin of [bind](http://underscorejs.org/#bind). You may pass \_ in your list of**arguments** to specify an argument that should not be pre-filled, but left open to supply at call-time.

var subtract = function(a, b) { return b - a; };

sub5 = \_.partial(subtract, 5);

sub5(20);

=> 15

// Using a placeholder

subFrom20 = \_.partial(subtract, \_, 20);

subFrom20(5);

=> 15

**memoize**\_.memoize(function, [hashFunction])   
Memoizes a given **function** by caching the computed result. Useful for speeding up slow-running computations. If passed an optional **hashFunction**, it will be used to compute the hash key for storing the result, based on the arguments to the original function. The default **hashFunction** just uses the first argument to the memoized function as the key. The cache of memoized values is available as the cache property on the returned function.

var fibonacci = \_.memoize(function(n) {

return n < 2 ? n: fibonacci(n - 1) + fibonacci(n - 2);

});

**delay**\_.delay(function, wait, \*arguments)   
Much like **setTimeout**, invokes **function** after **wait** milliseconds. If you pass the optional **arguments**, they will be forwarded on to the **function** when it is invoked.

var log = \_.bind(console.log, console);

\_.delay(log, 1000, 'logged later');

=> 'logged later' // Appears after one second.

**defer**\_.defer(function, \*arguments)   
Defers invoking the **function** until the current call stack has cleared, similar to using **setTimeout** with a delay of 0. Useful for performing expensive computations or HTML rendering in chunks without blocking the UI thread from updating. If you pass the optional **arguments**, they will be forwarded on to the **function** when it is invoked.

\_.defer(function(){ alert('deferred'); });

// Returns from the function before the alert runs.

**throttle**\_.throttle(function, wait, [options])   
Creates and returns a new, throttled version of the passed function, that, when invoked repeatedly, will only actually call the original function at most once per every **wait**milliseconds. Useful for rate-limiting events that occur faster than you can keep up with.

By default, **throttle** will execute the function as soon as you call it for the first time, and, if you call it again any number of times during the **wait** period, as soon as that period is over. If you'd like to disable the leading-edge call, pass {leading: false}, and if you'd like to disable the execution on the trailing-edge, pass   
{trailing: false}.

var throttled = \_.throttle(updatePosition, 100);

$(window).scroll(throttled);

**debounce**\_.debounce(function, wait, [immediate])   
Creates and returns a new debounced version of the passed function which will postpone its execution until after **wait** milliseconds have elapsed since the last time it was invoked. Useful for implementing behavior that should only happen *after* the input has stopped arriving. For example: rendering a preview of a Markdown comment, recalculating a layout after the window has stopped being resized, and so on.

At the end of the **wait** interval, the function will be called with the arguments that were passed *most recently* to the debounced function.

Pass true for the **immediate** argument to cause **debounce** to trigger the function on the leading instead of the trailing edge of the **wait** interval. Useful in circumstances like preventing accidental double-clicks on a "submit" button from firing a second time.

var lazyLayout = \_.debounce(calculateLayout, 300);

$(window).resize(lazyLayout);

**once**\_.once(function)   
Creates a version of the function that can only be called one time. Repeated calls to the modified function will have no effect, returning the value from the original call. Useful for initialization functions, instead of having to set a boolean flag and then check it later.

var initialize = \_.once(createApplication);

initialize();

initialize();

// Application is only created once.

**after**\_.after(count, function)   
Creates a version of the function that will only be run after being called **count** times. Useful for grouping asynchronous responses, where you want to be sure that all the async calls have finished, before proceeding.

var renderNotes = \_.after(notes.length, render);

\_.each(notes, function(note) {

note.asyncSave({success: renderNotes});

});

// renderNotes is run once, after all notes have saved.

**before**\_.before(count, function)   
Creates a version of the function that can be called no more than **count** times. The result of the last function call is memoized and returned when **count** has been reached.

var monthlyMeeting = \_.before(3, askForRaise);

monthlyMeeting();

monthlyMeeting();

monthlyMeeting();

// the result of any subsequent calls is the same as the second call

**wrap**\_.wrap(function, wrapper)   
Wraps the first **function** inside of the **wrapper** function, passing it as the first argument. This allows the **wrapper** to execute code before and after the **function**runs, adjust the arguments, and execute it conditionally.

var hello = function(name) { return "hello: " + name; };

hello = \_.wrap(hello, function(func) {

return "before, " + func("moe") + ", after";

});

hello();

=> 'before, hello: moe, after'

**negate**\_.negate(predicate)   
Returns a new negated version of the **predicate** function.

var isFalsy = \_.negate(Boolean);

\_.find([-2, -1, 0, 1, 2], isFalsy);

=> 0

**compose**\_.compose(\*functions)   
Returns the composition of a list of **functions**, where each function consumes the return value of the function that follows. In math terms, composing the functions *f()*, *g()*, and *h()* produces *f(g(h()))*.

var greet = function(name){ return "hi: " + name; };

var exclaim = function(statement){ return statement.toUpperCase() + "!"; };

var welcome = \_.compose(greet, exclaim);

welcome('moe');

=> 'hi: MOE!'

**Object Functions**

**keys**\_.keys(object)   
Retrieve all the names of the **object**'s own enumerable properties.

\_.keys({one: 1, two: 2, three: 3});

=> ["one", "two", "three"]

**allKeys**\_.allKeys(object)   
Retrieve *all* the names of **object**'s own and inherited properties.

function Stooge(name) {

this.name = name;

}

Stooge.prototype.silly = true;

\_.allKeys(new Stooge("Moe"));

=> ["name", "silly"]

**values**\_.values(object)   
Return all of the values of the **object**'s own properties.

\_.values({one: 1, two: 2, three: 3});

=> [1, 2, 3]

**mapObject**\_.mapObject(object, iteratee, [context])   
Like [map](http://underscorejs.org/#map), but for objects. Transform the value of each property in turn.

\_.mapObject({start: 5, end: 12}, function(val, key) {

return val + 5;

});

=> {start: 10, end: 17}

**pairs**\_.pairs(object)   
Convert an object into a list of [key, value] pairs.

\_.pairs({one: 1, two: 2, three: 3});

=> [["one", 1], ["two", 2], ["three", 3]]

**invert**\_.invert(object)   
Returns a copy of the **object** where the keys have become the values and the values the keys. For this to work, all of your object's values should be unique and string serializable.

\_.invert({Moe: "Moses", Larry: "Louis", Curly: "Jerome"});

=> {Moses: "Moe", Louis: "Larry", Jerome: "Curly"};

**create**\_.create(prototype, props)   
Creates a new object with the given prototype, optionally attaching **props** as *own* properties. Basically, Object.create, but without all of the property descriptor jazz.

var moe = \_.create(Stooge.prototype, {name: "Moe"});

**functions**\_.functions(object) *Alias:****methods***   
Returns a sorted list of the names of every method in an object — that is to say, the name of every function property of the object.

\_.functions(\_);

=> ["all", "any", "bind", "bindAll", "clone", "compact", "compose" ...

**findKey**\_.findKey(object, predicate, [context])   
Similar to [\_.findIndex](http://underscorejs.org/#findIndex) but for keys in objects. Returns the *key* where the **predicate**truth test passes or *undefined*.

**extend**\_.extend(destination, \*sources)   
Shallowly copy all of the properties **in** the **source** objects over to the **destination** object, and return the **destination** object. Any nested objects or arrays will be copied by reference, not duplicated. It's in-order, so the last source will override properties of the same name in previous arguments.

\_.extend({name: 'moe'}, {age: 50});

=> {name: 'moe', age: 50}

**extendOwn**\_.extendOwn(destination, \*sources) *Alias:****assign***   
Like **extend**, but only copies *own* properties over to the destination object.

**pick**\_.pick(object, \*keys)   
Return a copy of the **object**, filtered to only have values for the whitelisted **keys** (or array of valid keys). Alternatively accepts a predicate indicating which keys to pick.

\_.pick({name: 'moe', age: 50, userid: 'moe1'}, 'name', 'age');

=> {name: 'moe', age: 50}

\_.pick({name: 'moe', age: 50, userid: 'moe1'}, function(value, key, object) {

return \_.isNumber(value);

});

=> {age: 50}

**omit**\_.omit(object, \*keys)   
Return a copy of the **object**, filtered to omit the blacklisted **keys** (or array of keys). Alternatively accepts a predicate indicating which keys to omit.

\_.omit({name: 'moe', age: 50, userid: 'moe1'}, 'userid');

=> {name: 'moe', age: 50}

\_.omit({name: 'moe', age: 50, userid: 'moe1'}, function(value, key, object) {

return \_.isNumber(value);

});

=> {name: 'moe', userid: 'moe1'}

**defaults**\_.defaults(object, \*defaults)   
Fill in undefined properties in **object** with the first value present in the following list of **defaults** objects.

var iceCream = {flavor: "chocolate"};

\_.defaults(iceCream, {flavor: "vanilla", sprinkles: "lots"});

=> {flavor: "chocolate", sprinkles: "lots"}

**clone**\_.clone(object)   
Create a shallow-copied clone of the provided *plain* **object**. Any nested objects or arrays will be copied by reference, not duplicated.

\_.clone({name: 'moe'});

=> {name: 'moe'};

**tap**\_.tap(object, interceptor)   
Invokes **interceptor** with the **object**, and then returns **object**. The primary purpose of this method is to "tap into" a method chain, in order to perform operations on intermediate results within the chain.

\_.chain([1,2,3,200])

.filter(function(num) { return num % 2 == 0; })

.tap(alert)

.map(function(num) { return num \* num })

.value();

=> // [2, 200] (alerted)

=> [4, 40000]

**has**\_.has(object, key)   
Does the object contain the given key? Identical to object.hasOwnProperty(key), but uses a safe reference to the hasOwnProperty function, in case it's been [overridden accidentally](http://www.devthought.com/2012/01/18/an-object-is-not-a-hash/).

\_.has({a: 1, b: 2, c: 3}, "b");

=> true

**property**\_.property(key)   
Returns a function that will itself return the key property of any passed-in object.

var stooge = {name: 'moe'};

'moe' === \_.property('name')(stooge);

=> true

**propertyOf**\_.propertyOf(object)   
Inverse of \_.property. Takes an object and returns a function which will return the value of a provided property.

var stooge = {name: 'moe'};

\_.propertyOf(stooge)('name');

=> 'moe'

**matcher**\_.matcher(attrs) *Alias:****matches***   
Returns a predicate function that will tell you if a passed in object contains all of the key/value properties present in **attrs**.

var ready = \_.matcher({selected: true, visible: true});

var readyToGoList = \_.filter(list, ready);

**isEqual**\_.isEqual(object, other)   
Performs an optimized deep comparison between the two objects, to determine if they should be considered equal.

var stooge = {name: 'moe', luckyNumbers: [13, 27, 34]};

var clone = {name: 'moe', luckyNumbers: [13, 27, 34]};

stooge == clone;

=> false

\_.isEqual(stooge, clone);

=> true

**isMatch**\_.isMatch(object, properties)   
Tells you if the keys and values in **properties** are contained in **object**.

var stooge = {name: 'moe', age: 32};

\_.isMatch(stooge, {age: 32});

=> true

**isEmpty**\_.isEmpty(object)   
Returns *true* if an enumerable **object** contains no values (no enumerable own-properties). For strings and array-like objects \_.isEmpty checks if the length property is 0.

\_.isEmpty([1, 2, 3]);

=> false

\_.isEmpty({});

=> true

**isElement**\_.isElement(object)   
Returns *true* if **object** is a DOM element.

\_.isElement(jQuery('body')[0]);

=> true

**isArray**\_.isArray(object)   
Returns *true* if **object** is an Array.

(function(){ return \_.isArray(arguments); })();

=> false

\_.isArray([1,2,3]);

=> true

**isObject**\_.isObject(value)   
Returns *true* if **value** is an Object. Note that JavaScript arrays and functions are objects, while (normal) strings and numbers are not.

\_.isObject({});

=> true

\_.isObject(1);

=> false

**isArguments**\_.isArguments(object)   
Returns *true* if **object** is an Arguments object.

(function(){ return \_.isArguments(arguments); })(1, 2, 3);

=> true

\_.isArguments([1,2,3]);

=> false

**isFunction**\_.isFunction(object)   
Returns *true* if **object** is a Function.

\_.isFunction(alert);

=> true

**isString**\_.isString(object)   
Returns *true* if **object** is a String.

\_.isString("moe");

=> true

**isNumber**\_.isNumber(object)   
Returns *true* if **object** is a Number (including NaN).

\_.isNumber(8.4 \* 5);

=> true

**isFinite**\_.isFinite(object)   
Returns *true* if **object** is a finite Number.

\_.isFinite(-101);

=> true

\_.isFinite(-Infinity);

=> false

**isBoolean**\_.isBoolean(object)   
Returns *true* if **object** is either *true* or *false*.

\_.isBoolean(null);

=> false

**isDate**\_.isDate(object)   
Returns *true* if **object** is a Date.

\_.isDate(new Date());

=> true

**isRegExp**\_.isRegExp(object)   
Returns *true* if **object** is a RegExp.

\_.isRegExp(/moe/);

=> true

**isError**\_.isError(object)   
Returns *true* if **object** inherits from an Error.

try {

throw new TypeError("Example");

} catch (o\_O) {

\_.isError(o\_O);

}

=> true

**isNaN**\_.isNaN(object)   
Returns *true* if **object** is *NaN*.  
Note: this is not the same as the native **isNaN** function, which will also return true for many other not-number values, such as undefined.

\_.isNaN(NaN);

=> true

isNaN(undefined);

=> true

\_.isNaN(undefined);

=> false

**isNull**\_.isNull(object)   
Returns *true* if the value of **object** is *null*.

\_.isNull(null);

=> true

\_.isNull(undefined);

=> false

**isUndefined**\_.isUndefined(value)   
Returns *true* if **value** is *undefined*.

\_.isUndefined(window.missingVariable);

=> true

**Utility Functions**

**noConflict**\_.noConflict()   
Give control of the \_ variable back to its previous owner. Returns a reference to the **Underscore** object.

var underscore = \_.noConflict();

**identity**\_.identity(value)   
Returns the same value that is used as the argument. In math: f(x) = x  
This function looks useless, but is used throughout Underscore as a default iteratee.

var stooge = {name: 'moe'};

stooge === \_.identity(stooge);

=> true

**constant**\_.constant(value)   
Creates a function that returns the same value that is used as the argument of \_.constant.

var stooge = {name: 'moe'};

stooge === \_.constant(stooge)();

=> true

**noop**\_.noop()   
Returns undefined irrespective of the arguments passed to it. Useful as the default for optional callback arguments.

obj.initialize = \_.noop;

**times**\_.times(n, iteratee, [context])   
Invokes the given iteratee function **n** times. Each invocation of **iteratee** is called with an index argument. Produces an array of the returned values.   
*Note: this example uses the*[*object-oriented syntax*](http://underscorejs.org/#oop).

\_(3).times(function(n){ genie.grantWishNumber(n); });

**random**\_.random(min, max)   
Returns a random integer between **min** and **max**, inclusive. If you only pass one argument, it will return a number between 0 and that number.

\_.random(0, 100);

=> 42

**mixin**\_.mixin(object)   
Allows you to extend Underscore with your own utility functions. Pass a hash of {name: function} definitions to have your functions added to the Underscore object, as well as the OOP wrapper.

\_.mixin({

capitalize: function(string) {

return string.charAt(0).toUpperCase() + string.substring(1).toLowerCase();

}

});

\_("fabio").capitalize();

=> "Fabio"

**iteratee**\_.iteratee(value, [context])   
Generates a callback that can be applied to each element in a collection. \_.iterateesupports a number of shorthand syntaxes for common callback use cases. Depending upon value's type, \_.iteratee will return:

// No value

\_.iteratee();

=> \_.identity()

// Function

\_.iteratee(function(n) { return n \* 2; });

=> function(n) { return n \* 2; }

// Object

\_.iteratee({firstName: 'Chelsea'});

=> \_.matcher({firstName: 'Chelsea'});

// Anything else

\_.iteratee('firstName');

=> \_.property('firstName');

The following Underscore methods transform their predicates through \_.iteratee: countBy, every, filter, find, findIndex, findKey, findLastIndex, groupBy, indexBy, map, mapObject, max, min, partition, reject, some, sortBy,sortedIndex, and uniq

**uniqueId**\_.uniqueId([prefix])   
Generate a globally-unique id for client-side models or DOM elements that need one. If **prefix** is passed, the id will be appended to it.

\_.uniqueId('contact\_');

=> 'contact\_104'

**escape**\_.escape(string)   
Escapes a string for insertion into HTML, replacing &, <, >, ", `, and 'characters.

\_.escape('Curly, Larry & Moe');

=> "Curly, Larry &amp; Moe"

**unescape**\_.unescape(string)   
The opposite of [**escape**](http://underscorejs.org/#escape), replaces &amp;, &lt;, &gt;, &quot;, &#96; and &#x27;with their unescaped counterparts.

\_.unescape('Curly, Larry &amp; Moe');

=> "Curly, Larry & Moe"

**result**\_.result(object, property, [defaultValue])   
If the value of the named **property** is a function then invoke it with the **object** as context; otherwise, return it. If a default value is provided and the property doesn't exist or is undefined then the default will be returned. If defaultValue is a function its result will be returned.

var object = {cheese: 'crumpets', stuff: function(){ return 'nonsense'; }};

\_.result(object, 'cheese');

=> "crumpets"

\_.result(object, 'stuff');

=> "nonsense"

\_.result(object, 'meat', 'ham');

=> "ham"

**now**\_.now()   
Returns an integer timestamp for the current time, using the fastest method available in the runtime. Useful for implementing timing/animation functions.

\_.now();

=> 1392066795351

**template**\_.template(templateString, [settings])   
Compiles JavaScript templates into functions that can be evaluated for rendering. Useful for rendering complicated bits of HTML from JSON data sources. Template functions can both interpolate values, using <%= … %>, as well as execute arbitrary JavaScript code, with <% … %>. If you wish to interpolate a value, and have it be HTML-escaped, use <%- … %>. When you evaluate a template function, pass in a **data**object that has properties corresponding to the template's free variables. The **settings**argument should be a hash containing any \_.templateSettings that should be overridden.

var compiled = \_.template("hello: <%= name %>");

compiled({name: 'moe'});

=> "hello: moe"

var template = \_.template("<b><%- value %></b>");

template({value: '<script>'});

=> "<b>&lt;script&gt;</b>"

You can also use print from within JavaScript code. This is sometimes more convenient than using <%= ... %>.

var compiled = \_.template("<% print('Hello ' + epithet); %>");

compiled({epithet: "stooge"});

=> "Hello stooge"

If ERB-style delimiters aren't your cup of tea, you can change Underscore's template settings to use different symbols to set off interpolated code. Define an **interpolate**regex to match expressions that should be interpolated verbatim, an **escape** regex to match expressions that should be inserted after being HTML-escaped, and an **evaluate** regex to match expressions that should be evaluated without insertion into the resulting string. You may define or omit any combination of the three. For example, to perform [Mustache.js](https://github.com/janl/mustache.js#readme)-style templating:

\_.templateSettings = {

interpolate: /\{\{(.+?)\}\}/g

};

var template = \_.template("Hello {{ name }}!");

template({name: "Mustache"});

=> "Hello Mustache!"

By default, **template** places the values from your data in the local scope via the withstatement. However, you can specify a single variable name with the **variable** setting. This can significantly improve the speed at which a template is able to render.

\_.template("Using 'with': <%= data.answer %>", {variable: 'data'})({answer: 'no'});

=> "Using 'with': no"

Precompiling your templates can be a big help when debugging errors you can't reproduce. This is because precompiled templates can provide line numbers and a stack trace, something that is not possible when compiling templates on the client. The **source** property is available on the compiled template function for easy precompilation.

<script>

JST.project = <%= \_.template(jstText).source %>;

</script>

**Object-Oriented Style**

You can use Underscore in either an object-oriented or a functional style, depending on your preference. The following two lines of code are identical ways to double a list of numbers.

\_.map([1, 2, 3], function(n){ return n \* 2; });

\_([1, 2, 3]).map(function(n){ return n \* 2; });

**Chaining**

Calling chain will cause all future method calls to return wrapped objects. When you've finished the computation, call value to retrieve the final value. Here's an example of chaining together a **map/flatten/reduce**, in order to get the word count of every word in a song.

var lyrics = [

{line: 1, words: "I'm a lumberjack and I'm okay"},

{line: 2, words: "I sleep all night and I work all day"},

{line: 3, words: "He's a lumberjack and he's okay"},

{line: 4, words: "He sleeps all night and he works all day"}

];

\_.chain(lyrics)

.map(function(line) { return line.words.split(' '); })

.flatten()

.reduce(function(counts, word) {

counts[word] = (counts[word] || 0) + 1;

return counts;

}, {})

.value();

=> {lumberjack: 2, all: 4, night: 2 ... }

In addition, the [Array prototype's methods](https://developer.mozilla.org/en/JavaScript/Reference/Global_Objects/Array/prototype) are proxied through the chained Underscore object, so you can slip a reverse or a push into your chain, and continue to modify the array.

**chain**\_.chain(obj)   
Returns a wrapped object. Calling methods on this object will continue to return wrapped objects until value is called.

var stooges = [{name: 'curly', age: 25}, {name: 'moe', age: 21}, {name: 'larry', age: 23}];

var youngest = \_.chain(stooges)

.sortBy(function(stooge){ return stooge.age; })

.map(function(stooge){ return stooge.name + ' is ' + stooge.age; })

.first()

.value();

=> "moe is 21"

**value**\_.chain(obj).value()   
Extracts the value of a wrapped object.

\_.chain([1, 2, 3]).reverse().value();

=> [3, 2, 1]

**Links & Suggested Reading**

The Underscore documentation is also available in [Simplified Chinese](http://learning.github.io/underscore/).

[Underscore.lua](http://mirven.github.io/underscore.lua/), a Lua port of the functions that are applicable in both languages. Includes OOP-wrapping and chaining. ([source](https://github.com/mirven/underscore.lua))

[Dollar.swift](http://www.dollarswift.org/), a Swift port of many of the Underscore.js functions and more. ([source](https://github.com/ankurp/Dollar.swift))

[Underscore.m](http://underscorem.org/), an Objective-C port of many of the Underscore.js functions, using a syntax that encourages chaining. ([source](https://github.com/robb/Underscore.m))

[\_.m](http://kmalakoff.github.io/_.m/), an alternative Objective-C port that tries to stick a little closer to the original Underscore.js API. ([source](https://github.com/kmalakoff/_.m))

[Underscore.php](https://github.com/bdelespierre/underscore.php), a PHP port of the functions that are applicable in both languages. Tailored for PHP 5.4 and made with data-type tolerance in mind. ([source](https://github.com/bdelespierre/underscore.php))

[Underscore-perl](http://vti.github.io/underscore-perl/), a Perl port of many of the Underscore.js functions, aimed at on Perl hashes and arrays. ([source](https://github.com/vti/underscore-perl))

[Underscore.cfc](http://russplaysguitar.github.io/UnderscoreCF/), a Coldfusion port of many of the Underscore.js functions. ([source](https://github.com/russplaysguitar/underscorecf))

[Underscore.string](http://epeli.github.io/underscore.string/), an Underscore extension that adds functions for string-manipulation: trim, startsWith, contains, capitalize, reverse, sprintf, and more.

[Underscore-java](http://javadev.github.io/underscore-java/), a java port of the functions that are applicable in both languages. Includes OOP-wrapping and chaining. ([source](https://github.com/javadev/underscore-java))

Ruby's [Enumerable](http://ruby-doc.org/core/classes/Enumerable.html) module.

[Prototype.js](http://prototypejs.org/), which provides JavaScript with collection functions in the manner closest to Ruby's Enumerable.

Oliver Steele's [Functional JavaScript](http://osteele.com/sources/javascript/functional/), which includes comprehensive higher-order function support as well as string lambdas.

Michael Aufreiter's [Data.js](https://github.com/substance/data), a data manipulation + persistence library for JavaScript.

Python's [itertools](http://docs.python.org/library/itertools.html).

[PyToolz](http://toolz.readthedocs.org/), a Python port that extends itertools and functools to include much of the Underscore API.

[Funcy](https://github.com/Suor/funcy), a practical collection of functional helpers for Python, partially inspired by Underscore.