Underscore.js (1.8.3)

» GitHub Repository
» Annotated Source
» Underscore-contrib

Introduction

Collections

- each
- map
- reduce
- reduceRight
- find
- filter
- where
- findWhere
- reject
- every
- some
- contains
- invoke
- pluck
- max
- min
- sortBy
- groupBy
- indexBy
- countBy
- shuffle
- sample
- toArray
- size
- partition

Arrays

- first
- initial
- last
- rest
- compact
- flatten
- without
- union
- intersection
- difference
- uniq
- zip
- unzip
- object
- indexOf
- lastIndexOf
- sortedIndex
- findIndex
- findLastIndex
- range

Functions

- bind
- bindAll
- partial
- memoize
- delay
- defer
- throttle
- debounce
- once
- after
- before
- wrap

UNDERSCORE.JS

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's tux and Backbone'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 is included for your perusal.

You may also read through the annotated source code.

Enjoying Underscore, and want to *turn it up to 11?* Try Underscore-contrib.

The project is hosted on GitHub. You can report bugs and discuss features on the issues page, or on Freenode in the `#documentcloud` channel.

Underscore is an open-source component of DocumentCloud.

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

Development Version (1.8.3)	*52kb, Uncompressed with Plentiful Comments*
Production Version (1.8.3)	*5.7kb, Minified and Gzipped* (Source Map)
Edge Version	*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...
```

- negate
- compose

Objects

- keys
- allKeys
- values
- mapObject
- pairs
- invert
- create
- functions
- findKey
- extend
- extendOwn
- pick
- omit
- defaults
- clone
- tap
- has
- matcher
- property
- propertyOf
- isEqual
- isMatch
- isEmpty
- isElement
- isArray
- isObject
- isArguments
- isFunction
- isString
- isNumber
- isFinite
- isBoolean
- isDate
- isRegExp
- isNaN
- isNull
- isUndefined

Utility

- noConflict
- identity
- constant
- noop
- times
- random
- mixin
- iteratee
- uniqueId
- escape
- unescape
- result
- now
- template

Chaining

- chain
- value

Links

Change Log

```
_.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 entire `list`.

```
_.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 `value` and `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**. Delegates to the JavaScript 1.8 version of **reduceRight**, if it exists. **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.

```
_.every([true, 1, null, 'yes'], _.identity);
=> 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 guard may be required.

```
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 guard may be required.

```
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, 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.

```
_.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])` *Alias: **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])` *Alias: **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. If you're working with a matrix of nested arrays, `_.zip.apply` can transpose the matrix in a similar fashion.

```
_.zip(['moe', 'larry', 'curly'], [30, 40, 50], [true, false, false]);
=> [["moe", 30, true], ["larry", 40, false], ["curly", 50, false]]
```

unzip `_.unzip(*arrays)`

The opposite of zip. Given a number 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. Use with `apply` to pass in an array of arrays.

```
_.unzip([['moe', 'larry', 'curly'], [30, 40, 50], [true, false, false]])
=> ["moe", 30, true], ["larry", 40, false], ["curly", 50, 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`, 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` 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.

```
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').bind('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. 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.

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 first 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, 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` but for keys in objects. Returns the *key* where the **predicate** truth test passes or *undefined*.

extend `_.extend(destination, *sources)`

Copy all of the properties **in** the **source** objects over to the **destination** object, and return the **destination** object. 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.

```
_.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** inherrits 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 chaining syntax.

```
_(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])`

A mostly-internal function to generate callbacks that can be applied to each element in a collection, returning the desired result — either identity, an arbitrary callback, a property matcher, or a property accessor.
The full list of Underscore methods that transform predicates through `_.iteratee` is `map`, `find`, `filter`, `reject`, `every`, `some`, `max`, `min`, `sortBy`, `groupBy`, `indexBy`, `countBy`, `sortedIndex`, `partition`, and `unique`.

```
var stooges = [{name: 'curly', age: 25}, {name: 'moe', age: 21}, {name: 'larry', age: 23}];
_.map(stooges, _.iteratee('age'));
=> [25, 21, 23];
```

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**, replaces `&`, `<`, `>`, `&quot;`, ``` 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-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 `with` statement. 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>
```

Chaining

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; });
```

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 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 `_(obj).value()`

Extracts the value of a wrapped object.

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

Links & Suggested Reading

The Underscore documentation is also available in Simplified Chinese.

Underscore.lua, a Lua port of the functions that are applicable in both languages. Includes OOP-wrapping and chaining. (source)

Dollar.swift, a Swift port of many of the Underscore.js functions and more. (source)

Underscore.m, an Objective-C port of many of the Underscore.js functions, using a syntax that encourages chaining. (source)

_.m, an alternative Objective-C port that tries to stick a little closer to the original Underscore.js API. (source)

Underscore.php, a PHP port of the functions that are applicable in both languages. Includes OOP-wrapping and chaining. (source)

Underscore-perl, a Perl port of many of the Underscore.js functions, aimed at on Perl hashes and arrays. (source)

Underscore.cfc, a Coldfusion port of many of the Underscore.js functions. (source)

Underscore.string, an Underscore extension that adds functions for string-manipulation: `trim`, `startsWith`, `contains`, `capitalize`, `reverse`, `sprintf`, and more.

Underscore-java, a java port of the functions that are applicable in both languages. Includes OOP-wrapping and chaining. (source)

Ruby's Enumerable module.

Prototype.js, which provides JavaScript with collection functions in the manner closest to Ruby's Enumerable.

Oliver Steele's Functional JavaScript, which includes comprehensive higher-order function support as well as string lambdas.

Michael Aufreiter's Data.js, a data manipulation + persistence library for JavaScript.

Python's itertools.

PyToolz, a Python port that extends itertools and functools to include much of the Underscore API.

Funcy, a practical collection of functional helpers for Python, partially inspired by Underscore.

Change Log

1.8.3 — *April 2, 2015* — Diff — Docs

- Adds an `_.create` method, as a slimmed down version of `Object.create`.
- Works around an iOS bug that can improperly cause `isArrayLike` to be JIT-ed. Also fixes a bug when passing `0` to `isArrayLike`.

1.8.2 — *Feb. 22, 2015* — Diff — Docs

- Restores the previous old-Internet-Explorer edge cases changed in 1.8.1.
- Adds a `fromIndex` argument to `_.contains`.

1.8.1 — *Feb. 19, 2015* — Diff — Docs

- Fixes/changes some old-Internet Explorer and related edge case behavior. Test your app with Underscore 1.8.1 in an old IE and let us know how it's doing...

1.8.0 — *Feb. 19, 2015* — Diff — Docs

- Added `_.mapObject`, which is similar to `_.map`, but just for the values in your object. (A real crowd pleaser.)
- Added `_.allKeys` which returns *all* the enumerable property names on an object.
- Reverted a 1.7.0 change where `_.extend` only copied "own" properties. Hopefully

this will un-break you — if it breaks you again, I apologize.

- Added `_.extendOwn` — a less-useful form of `_.extend` that only copies over "own" properties.
- Added `_.findIndex` and `_.findLastIndex` functions, which nicely complement their twin-twins `_.indexOf` and `_.lastIndexOf`.
- Added an `_.isMatch` predicate function that tells you if an object matches key-value properties. A kissing cousin of `_.isEqual` and `_.matcher`.
- Added an `_.isError` function.
- Restored the `_.unzip` function as the inverse of `zip`. Flip-flopping. I know.
- `_.result` now takes an optional fallback value (or function that provides the fallback value).
- Added the `_.propertyOf` function generator as a mirror-world version of `_.property`.
- Deprecated `_.matches`. It's now known by a more harmonious name — `_.matcher`.
- Various and diverse code simplifications, changes for improved cross-platform compatibility, and edge case bug fixes.

1.7.0 — *August 26, 2014* — Diff — Docs

- For consistency and speed across browsers, Underscore now ignores native array methods for `forEach`, `map`, `reduce`, `reduceRight`, `filter`, `every`, `some`, `indexOf`, and `lastIndexOf`. "Sparse" arrays are officially dead in Underscore.
- Added `_.iteratee` to customize the iterators used by collection functions. Many Underscore methods will take a string argument for easier `_.property`-style lookups, an object for `_.where`-style filtering, or a function as a custom callback.
- Added `_.before` as a counterpart to `_.after`.
- Added `_.negate` to invert the truth value of a passed-in predicate.
- Added `_.noop` as a handy empty placeholder function.
- `_.isEmpty` now works with `arguments` objects.
- `_.has` now guards against nullish objects.
- `_.omit` can now take an iteratee function.
- `_.partition` is now called with `index` and `object`.
- `_.matches` creates a shallow clone of your object and only iterates over own properties.
- Aligning better with the forthcoming ECMA6 `Object.assign`, `_.extend` only iterates over the object's own properties.
- Falsey guards are no longer needed in `_.extend` and `_.defaults` —if the passed in argument isn't a JavaScript object it's just returned.
- Fixed a few edge cases in `_.max` and `_.min` to handle arrays containing `NaN` (like strings or other objects) and `Infinity` and `-Infinity`.
- Override base methods like `each` and `some` and they'll be used internally by other Underscore functions too.
- The escape functions handle backticks (`), to deal with an IE ≤ 8 bug.
- For consistency, `_.union` and `_.difference` now only work with arrays and not

variadic args.

- `_.memoize` exposes the cache of memoized values as a property on the returned function.
- `_.pick` accepts `iteratee` and `context` arguments for a more advanced callback.
- Underscore templates no longer accept an initial `data` object. `_.template` always returns a function now.
- Optimizations and code cleanup aplenty.

1.6.0 — *February 10, 2014* — Diff — Docs

- Underscore now registers itself for AMD (Require.js), Bower and Component, as well as being a CommonJS module and a regular (Java)Script. An ugliness, but perhaps a necessary one.
- Added `_.partition`, a way to split a collection into two lists of results — those that pass and those that fail a particular predicate.
- Added `_.property`, for easy creation of iterators that pull specific properties from objects. Useful in conjunction with other Underscore collection functions.
- Added `_.matches`, a function that will give you a predicate that can be used to tell if a given object matches a list of specified key/value properties.
- Added `_.constant`, as a higher-order `_.identity`.
- Added `_.now`, an optimized way to get a timestamp — used internally to speed up `debounce` and `throttle`.
- The `_.partial` function may now be used to partially apply any of its arguments, by passing `_` wherever you'd like a placeholder variable, to be filled-in later.
- The `_.each` function now returns a reference to the list for chaining.
- The `_.keys` function now returns an empty array for non-objects instead of throwing.
- … and more miscellaneous refactoring.

1.5.2 — *September 7, 2013* — Diff — Docs

- Added an `indexBy` function, which fits in alongside its cousins, `countBy` and `groupBy`.
- Added a `sample` function, for sampling random elements from arrays.
- Some optimizations relating to functions that can be implemented in terms of `_.keys` (which includes, significantly, `each` on objects). Also for `debounce` in a tight loop.
- The `_.escape` function no longer escapes '/'.

1.5.1 — *July 8, 2013* — Diff — Docs

- Removed `unzip`, as it's simply the application of `zip` to an array of arguments. Use `_.zip.apply(_, list)` to transpose instead.

1.5.0 — *July 6, 2013* — Diff — Docs

- Added a new `unzip` function, as the inverse of `_.zip`.

- The `throttle` function now takes an `options` argument, allowing you to disable execution of the throttled function on either the **leading** or **trailing** edge.
- A source map is now supplied for easier debugging of the minified production build of Underscore.
- The `defaults` function now only overrides `undefined` values, not `null` ones.
- Removed the ability to call `_.bindAll` with no method name arguments. It's pretty much always wiser to white-list the names of the methods you'd like to bind.
- Removed the ability to call `_.after` with an invocation count of zero. The minimum number of calls is (naturally) now `1`.

1.4.4 — *January 30, 2013* — Diff — Docs

- Added `_.findWhere`, for finding the first element in a list that matches a particular set of keys and values.
- Added `_.partial`, for partially applying a function *without* changing its dynamic reference to `this`.
- Simplified `bind` by removing some edge cases involving constructor functions. In short: don't `_.bind` your constructors.
- A minor optimization to `invoke`.
- Fix bug in the minified version due to the minifier incorrectly optimizing-away `isFunction`.

1.4.3 — *December 4, 2012* — Diff — Docs

- Improved Underscore compatibility with Adobe's JS engine that can be used to script Illustrator, Photoshop, and friends.
- Added a default `_.identity` iterator to `countBy` and `groupBy`.
- The `uniq` function can now take `array, iterator, context` as the argument list.
- The `times` function now returns the mapped array of iterator results.
- Simplified and fixed bugs in `throttle`.

1.4.2 — *October 6, 2012* — Diff — Docs

- For backwards compatibility, returned to pre-1.4.0 behavior when passing `null` to iteration functions. They now become no-ops again.

1.4.1 — *October 1, 2012* — Diff — Docs

- Fixed a 1.4.0 regression in the `lastIndexOf` function.

1.4.0 — *September 27, 2012* — Diff — Docs

- Added a `pairs` function, for turning a JavaScript object into `[key, value]` pairs ... as well as an `object` function, for converting an array of `[key, value]` pairs into an object.
- Added a `countBy` function, for counting the number of objects in a list that match a certain criteria.
- Added an `invert` function, for performing a simple inversion of the keys and values in an object.

- Added a `where` function, for easy cases of filtering a list for objects with specific values.
- Added an `omit` function, for filtering an object to remove certain keys.
- Added a `random` function, to return a random number in a given range.
- `_.debounce`'d functions now return their last updated value, just like `_.throttle`'d functions do.
- The `sortBy` function now runs a stable sort algorithm.
- Added the optional `fromIndex` option to `indexOf` and `lastIndexOf`.
- "Sparse" arrays are no longer supported in Underscore iteration functions. Use a `for` loop instead (or better yet, an object).
- The `min` and `max` functions may now be called on *very* large arrays.
- Interpolation in templates now represents `null` and `undefined` as the empty string.
- ~~Underscore iteration functions no longer accept `null` values as a no-op argument. You'll get an early error instead.~~
- A number of edge-cases fixes and tweaks, which you can spot in the diff. Depending on how you're using Underscore, **1.4.0** may be more backwards-incompatible than usual — please test when you upgrade.

1.3.3 — *April 10, 2012* — Diff — Docs

- Many improvements to `_.template`, which now provides the `source` of the template function as a property, for potentially even more efficient pre-compilation on the server-side. You may now also set the `variable` option when creating a template, which will cause your passed-in data to be made available under the variable you named, instead of using a `with` statement — significantly improving the speed of rendering the template.
- Added the `pick` function, which allows you to filter an object literal with a whitelist of allowed property names.
- Added the `result` function, for convenience when working with APIs that allow either functions or raw properties.
- Added the `isFinite` function, because sometimes knowing that a value is a number just ain't quite enough.
- The `sortBy` function may now also be passed the string name of a property to use as the sort order on each object.
- Fixed `uniq` to work with sparse arrays.
- The `difference` function now performs a shallow flatten instead of a deep one when computing array differences.
- The `debounce` function now takes an `immediate` parameter, which will cause the callback to fire on the leading instead of the trailing edge.

1.3.1 — *January 23, 2012* — Diff — Docs

- Added an `_.has` function, as a safer way to use `hasOwnProperty`.
- Added `_.collect` as an alias for `_.map`. Smalltalkers, rejoice.
- Reverted an old change so that `_.extend` will correctly copy over keys with undefined values again.

- Bugfix to stop escaping slashes within interpolations in `_.template`.

1.3.0 — *January 11, 2012* — Diff — Docs

- Removed AMD (RequireJS) support from Underscore. If you'd like to use Underscore with RequireJS, you can load it as a normal script, wrap or patch your copy, or download a forked version.

1.2.4 — *January 4, 2012* — Diff — Docs

- You now can (and probably should, as it's simpler) write `_.chain(list)` instead of `_(list).chain()`.
- Fix for escaped characters in Underscore templates, and for supporting customizations of `_.templateSettings` that only define one or two of the required regexes.
- Fix for passing an array as the first argument to an `_.wrap`'d function.
- Improved compatibility with ClojureScript, which adds a `call` function to `String.prototype`.

1.2.3 — *December 7, 2011* — Diff — Docs

- Dynamic scope is now preserved for compiled `_.template` functions, so you can use the value of `this` if you like.
- Sparse array support of `_.indexOf`, `_.lastIndexOf`.
- Both `_.reduce` and `_.reduceRight` can now be passed an explicitly `undefined` value. (There's no reason why you'd want to do this.)

1.2.2 — *November 14, 2011* — Diff — Docs

- Continued tweaks to `_.isEqual` semantics. Now JS primitives are considered equivalent to their wrapped versions, and arrays are compared by their numeric properties only (#351).
- `_.escape` no longer tries to be smart about not double-escaping already-escaped HTML entities. Now it just escapes regardless (#350).
- In `_.template`, you may now leave semicolons out of evaluated statements if you wish: `<% }) %>` (#369).
- `_.after(callback, 0)` will now trigger the callback immediately, making "after" easier to use with asynchronous APIs (#366).

1.2.1 — *October 24, 2011* — Diff — Docs

- Several important bug fixes for `_.isEqual`, which should now do better on mutated Arrays, and on non-Array objects with `length` properties. (#329)
- James Burke contributed Underscore exporting for AMD module loaders, and Tony Lukasavage for Appcelerator Titanium. (#335, #338)
- You can now `_.groupBy(list, 'property')` as a shortcut for grouping values by a particular common property.
- `_.throttle`'d functions now fire immediately upon invocation, and are rate-limited thereafter (#170, #266).

- Most of the `_.is[Type]` checks no longer ducktype.
- The `_.bind` function now also works on constructors, a-la ES5 ... but you would never want to use `_.bind` on a constructor function.
- `_.clone` no longer wraps non-object types in Objects.
- `_.find` and `_.filter` are now the preferred names for `_.detect` and `_.select`.

1.2.0 — *October 5, 2011* — Diff — Docs

- The `_.isEqual` function now supports true deep equality comparisons, with checks for cyclic structures, thanks to Kit Cambridge.
- Underscore templates now support HTML escaping interpolations, using `<%- ... %>` syntax.
- Ryan Tenney contributed `_.shuffle`, which uses a modified Fisher-Yates to give you a shuffled copy of an array.
- `_.uniq` can now be passed an optional iterator, to determine by what criteria an object should be considered unique.
- `_.last` now takes an optional argument which will return the last N elements of the list.
- A new `_.initial` function was added, as a mirror of `_.rest`, which returns all the initial values of a list (except the last N).

1.1.7 — *July 13, 2011* — Diff — Docs

Added `_.groupBy`, which aggregates a collection into groups of like items. Added `_.union` and `_.difference`, to complement the (re-named) `_.intersection`. Various improvements for support of sparse arrays. `_.toArray` now returns a clone, if directly passed an array. `_.functions` now also returns the names of functions that are present in the prototype chain.

1.1.6 — *April 18, 2011* — Diff — Docs

Added `_.after`, which will return a function that only runs after first being called a specified number of times. `_.invoke` can now take a direct function reference. `_.every` now requires an iterator function to be passed, which mirrors the ES5 API. `_.extend` no longer copies keys when the value is undefined. `_.bind` now errors when trying to bind an undefined value.

1.1.5 — *March 20, 2011* — Diff — Docs

Added an `_.defaults` function, for use merging together JS objects representing default options. Added an `_.once` function, for manufacturing functions that should only ever execute a single time. `_.bind` now delegates to the native ES5 version, where available. `_.keys` now throws an error when used on non-Object values, as in ES5. Fixed a bug with `_.keys` when used over sparse arrays.

1.1.4 — *January 9, 2011* — Diff — Docs

Improved compliance with ES5's Array methods when passing `null` as a value. `_.wrap` now correctly sets `this` for the wrapped function. `_.indexOf` now takes an optional flag for finding the insertion index in an array that is guaranteed to already be sorted. Avoiding the use of `.callee`, to allow `_.isArray` to work properly in ES5's strict mode.

1.1.3 — *December 1, 2010* — Diff — Docs

In CommonJS, Underscore may now be required with just:

`var _ = require("underscore")`. Added `_.throttle` and `_.debounce` functions. Removed `_.breakLoop`, in favor of an ES5-style un-*break*-able each implementation — this removes the try/catch, and you'll now have better stack traces for exceptions that are thrown within an Underscore iterator. Improved the **isType** family of functions for better interoperability with Internet Explorer host objects. `_.template` now correctly escapes backslashes in templates. Improved `_.reduce` compatibility with the ES5 version: if you don't pass an initial value, the first item in the collection is used. `_.each` no longer returns the iterated collection, for improved consistency with ES5's `forEach`.

1.1.2 — *October 15, 2010* — Diff — Docs

Fixed `_.contains`, which was mistakenly pointing at `_.intersect` instead of `_.include`, like it should have been. Added `_.unique` as an alias for `_.uniq`.

1.1.1 — *October 5, 2010* — Diff — Docs

Improved the speed of `_.template`, and its handling of multiline interpolations. Ryan Tenney contributed optimizations to many Underscore functions. An annotated version of the source code is now available.

1.1.0 — *August 18, 2010* — Diff — Docs

The method signature of `_.reduce` has been changed to match the ES5 signature, instead of the Ruby/Prototype.js version. This is a backwards-incompatible change. `_.template` may now be called with no arguments, and preserves whitespace. `_.contains` is a new alias for `_.include`.

1.0.4 — *June 22, 2010* — Diff — Docs

Andri Möll contributed the `_.memoize` function, which can be used to speed up expensive repeated computations by caching the results.

1.0.3 — *June 14, 2010* — Diff — Docs

Patch that makes `_.isEqual` return `false` if any property of the compared object has a `NaN` value. Technically the correct thing to do, but of questionable semantics. Watch out for NaN comparisons.

1.0.2 — *March 23, 2010* — Diff — Docs

Fixes `_.isArguments` in recent versions of Opera, which have arguments objects as real Arrays.

1.0.1 — *March 19, 2010* — Diff — Docs

Bugfix for `_.isEqual`, when comparing two objects with the same number of undefined keys, but with different names.

1.0.0 — *March 18, 2010* — Diff — Docs

Things have been stable for many months now, so Underscore is now considered to be out of beta, at **1.0**. Improvements since **0.6** include `_.isBoolean`, and the ability to have `_.extend` take multiple source objects.

0.6.0 — *February 24, 2010* — Diff — Docs

Major release. Incorporates a number of Mile Frawley's refactors for safer duck-typing on collection functions, and cleaner internals. A new `_.mixin` method that allows you to extend Underscore with utility functions of your own. Added `_.times`, which works the same as in Ruby or Prototype.js. Native support for ES5's `Array.isArray`, and `Object.keys`.

0.5.8 — *January 28, 2010* — Diff — Docs

Fixed Underscore's collection functions to work on NodeLists and HTMLCollections once more, thanks to Justin Tulloss.

0.5.7 — *January 20, 2010* — Diff — Docs

A safer implementation of `_.isArguments`, and a faster `_.isNumber`, thanks to Jed Schmidt.

0.5.6 — *January 18, 2010* — Diff — Docs

Customizable delimiters for `_.template`, contributed by Noah Sloan.

0.5.5 — *January 9, 2010* — Diff — Docs

Fix for a bug in MobileSafari's OOP-wrapper, with the arguments object.

0.5.4 — *January 5, 2010* — Diff — Docs

Fix for multiple single quotes within a template string for `_.template`. See: Rick Strahl's blog post.

0.5.2 — *January 1, 2010* — Diff — Docs

New implementations of `isArray`, `isDate`, `isFunction`, `isNumber`, `isRegExp`, and `isString`, thanks to a suggestion from Robert Kieffer. Instead of doing `Object#toString` comparisons, they now check for expected properties, which is less safe, but more than an order of magnitude faster. Most other Underscore functions saw minor speed improvements as a result. Evgeniy Dolzhenko contributed `_.tap`, similar to Ruby 1.9's, which is handy for injecting side effects (like logging) into chained calls.

0.5.1 — *December 9, 2009* — Diff — Docs

Added an `_.isArguments` function. Lots of little safety checks and optimizations contributed by Noah Sloan and Andri Möll.

0.5.0 — *December 7, 2009* — Diff — Docs

[API Changes] `_.bindAll` now takes the context object as its first parameter. If no method names are passed, all of the context object's methods are bound to it, enabling chaining and easier binding. `_.functions` now takes a single argument and returns the names of its Function properties. Calling `_.functions(_)` will get you the previous behavior. Added `_.isRegExp` so that `isEqual` can now test for RegExp equality. All of the "is" functions have been shrunk down into a single definition. Karl Guertin contributed patches.

0.4.7 — *December 6, 2009* — Diff — Docs

Added `isDate`, `isNaN`, and `isNull`, for completeness. Optimizations for `isEqual` when checking equality between Arrays or Dates. `_.keys` is now ***25%–2X*** faster (depending on your browser) which speeds up the functions that rely on it, such as `_.each`.

0.4.6 — *November 30, 2009* — Diff — Docs

Added the `range` function, a port of the Python function of the same name, for generating flexibly-numbered lists of integers. Original patch contributed by Kirill Ishanov.

0.4.5 — *November 19, 2009* — Diff — Docs

Added `rest` for Arrays and arguments objects, and aliased `first` as `head`, and `rest` as `tail`, thanks to Luke Sutton's patches. Added tests ensuring that all

Underscore Array functions also work on *arguments* objects.

0.4.4 — *November 18, 2009* — Diff — Docs

Added `isString`, and `isNumber`, for consistency. Fixed `_.isEqual(NaN, NaN)` to return *true* (which is debatable).

0.4.3 — *November 9, 2009* — Diff — Docs

Started using the native `StopIteration` object in browsers that support it. Fixed Underscore setup for CommonJS environments.

0.4.2 — *November 9, 2009* — Diff — Docs

Renamed the unwrapping function to `value`, for clarity.

0.4.1 — *November 8, 2009* — Diff — Docs

Chained Underscore objects now support the Array prototype methods, so that you can perform the full range of operations on a wrapped array without having to break your chain. Added a `breakLoop` method to **break** in the middle of any Underscore iteration. Added an `isEmpty` function that works on arrays and objects.

0.4.0 — *November 7, 2009* — Diff — Docs

All Underscore functions can now be called in an object-oriented style, like so: `_([1, 2, 3]).map(...);`. Original patch provided by Marc-André Cournoyer. Wrapped objects can be chained through multiple method invocations. A `functions` method was added, providing a sorted list of all the functions in Underscore.

0.3.3 — *October 31, 2009* — Diff — Docs

Added the JavaScript 1.8 function `reduceRight`. Aliased it as `foldr`, and aliased `reduce` as `foldl`.

0.3.2 — *October 29, 2009* — Diff — Docs

Now runs on stock Rhino interpreters with: `load("underscore.js")`. Added `identity` as a utility function.

0.3.1 — *October 29, 2009* — Diff — Docs

All iterators are now passed in the original collection as their third argument, the same as JavaScript 1.6's **forEach**. Iterating over objects is now called with `(value, key, collection)`, for details see `_.each`.

0.3.0 — *October 29, 2009* — Diff — Docs

Added Dmitry Baranovskiy's comprehensive optimizations, merged in Kris Kowal's patches to make Underscore CommonJS and Narwhal compliant.

0.2.0 — *October 28, 2009* — Diff — Docs

Added `compose` and `lastIndexOf`, renamed `inject` to `reduce`, added aliases for `inject`, `filter`, `every`, `some`, and `forEach`.

0.1.1 — *October 28, 2009* — Diff — Docs

Added `noConflict`, so that the "Underscore" object can be assigned to other variables.

0.1.0 — *October 28, 2009* — Docs

Initial release of Underscore.js.


A DocumentCloud Project