Underscore Reference — *Smooth CoffeeScript*

This reference is an adaptation of the documentation found at Underscore.js. It is *interactive* in its HTML form. Edit a CoffeeScript segment to try it. You can see the generated JavaScript when you write a CoffeeScript function by typing ‘show name’ after its definition.

```
if exports?
  _ = require 'underscore'
else
  _ = window._ # Workaround for interactive environment quirk.

view = (obj) ->
  show if typeof obj is 'object'
    """{#{"\n  #{k}: #{v}" for own k,v of obj}\n}"""
  else obj

tryIt = ->
  show view # Show equivalent JavaScript
  view {
    'JavaScript' : "we could have been the closest of friends"
    'EcmaScript' : "we might have been the world's greatest lovers"
    'But'        : "now we're just without each other"
  }
# Uncomment the next line to try it
# tryIt()
```

Underscore version 1.2.3

Underscore is a library for functional style programming. It provides 60-odd functions that support both the usual functional suspects: **map**, **select**, **invoke** — as well as more specialized helpers: function binding, javascript templating, deep equality testing, and so on. It delegates to built-in functions, if present, so modern browsers will use the native implementations of **forEach**, **map**, **reduce**, **filter**, **every**, **some** and **indexOf**.

Underscore is an open-source component of DocumentCloud. You can find more information and updates at Underscore.js.

Downloads

Right-click, and use "Save As"

- Development Version
  - *34kb, Uncompressed with Comments*
- Production Version
  - *< 4kb, Minified and Gzipped*

Object-Oriented and Functional Styles

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.

```
show _.map [ 1, 2, 3 ], (n) -> n * 2
show _([ 1, 2, 3 ]).map (n) -> n * 2
```

Using the object-oriented style allows you to chain together methods. Calling `chain` on a wrapped object will cause all future method calls to return wrapped objects as well. When you've finished the computation, use `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.

```
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"}
]
view _(lyrics).chain()
  .map((line) -> line.words.split " ")
  .flatten()
  .reduce(((counts, word) ->
    counts[word] = (counts[word] or 0) + 1
    counts), {})
  .value()
```

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.

Collection Functions (Arrays or Objects)

each `_.each list, iterator, [context]` Alias: **forEach**

Iterates over a **list** of elements, yielding each in turn to an **iterator** function. The **iterator** is bound to the **context** object, if one is passed. Each invocation of **iterator** is called with three arguments: `element, index, list`. If **list** is a JavaScript object, **iterator**'s arguments will be `value, key, list`. Delegates to the native **forEach** function if it exists.

```
_.each [ 1, 2, 3 ], (num) -> show num
```

```
_.each {one : 1, two : 2, three : 3}, (num, key) -> show num
```

map `_.map list, iterator, [context]`

Produces a new array of values by mapping each value in **list** through a transformation function (**iterator**). If the native **map** method exists, it will be used instead. If **list** is a JavaScript object, **iterator**'s arguments will be `value, key, list`.

```
show _.map [ 1, 2, 3 ], (num) -> num * 3
```

```
show _.map
  one: 1
  two: 2
  three: 3
, (num, key) ->
  num * 3
```

reduce `_.reduce list, iterator, 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 **iterator**.

```
show sum = _.reduce [1, 2, 3], ((memo, num) -> memo + num), 0
```

reduceRight `_.reduceRight list, iterator, 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.

```
list = [ [ 0, 1 ], [ 2, 3 ], [ 4, 5 ] ]
flat = _.reduceRight list, (a, b) ->
  a.concat b
, []
show flat
```

find `_.find list, iterator, [context]` Alias: **detect**

Looks through each value in the **list**, returning the first one that passes a truth test (**iterator**). The function returns as soon as it finds an acceptable element, and doesn't traverse the entire list.

```
show even = _.find [1..6], (num) -> num % 2 is 0
```

filter `_.filter list, iterator, [context]` Alias: **select**

Looks through each value in the **list**, returning an array of all the values that pass a truth test (**iterator**). Delegates to the native **filter** method, if it exists.

```
show evens = _.filter [1..6], (num) -> num % 2 is 0
```

reject `_.reject list, iterator, [context]`

Returns the values in **list** without the elements that the truth test (**iterator**) passes. The opposite of **filter**.

```
show odds = _.reject [1..6], (num) -> num % 2 is 0
```

all `_.all list, iterator, [context]` Alias: **every**

Returns *true* if all of the values in the **list** pass the **iterator** truth test. Delegates to the native method **every**, if present.

```
show _.all [true, 1, null, 'yes'], _.identity
```

any `_.any list, [iterator], [context]` Alias: **some**

Returns *true* if any of the values in the **list** pass the **iterator** truth test. Short-circuits and stops traversing the list if a true element is found. Delegates to the native method **some**, if present.

```
show _.any [null, 0, 'yes', false]
```

include `_.include list, value` Alias: **contains**

Returns *true* if the **value** is present in the **list**, using === to test equality. Uses **indexOf** internally, if **list** is an Array.

```
show _.include [1, 2, 3], 3
```

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.

```
show _.invoke [[5, 1, 7], [3, 2, 1]], 'sort'
```

pluck `_.pluck list, propertyName`

A convenient version of what is perhaps the most common use-case for **map**: extracting a list of property values.

```
stooges = [
  {name : 'moe', age : 40}
  {name : 'larry', age : 50}
  {name : 'curly', age : 60}
]
show _.pluck stooges, 'name'
```

max `_.max list, [iterator], [context]`

Returns the maximum value in **list**. If **iterator** is passed, it will be used on each value to generate the criterion by which the value is ranked.

```
stooges = [
  {name : 'moe', age : 40}
  {name : 'larry', age : 50}
  {name : 'curly', age : 60}
]
view _.max stooges, (stooge) -> stooge.age
```

min `_.min list, [iterator], [context]`

Returns the minimum value in **list**. If **iterator** is passed, it will be used on each value to generate the criterion by which the value is ranked.

```
numbers = [10, 5, 100, 2, 1000]
show _.min numbers
```

sortBy `_.sortBy list, iterator, [context]`

Returns a sorted copy of **list**, ranked by the results of running each value through **iterator**.

```
show _.sortBy [1..6], (num) -> Math.sin num
```

groupBy `_.groupBy list, iterator`

Splits a collection into sets, grouped by the result of running each value through **iterator**. If **iterator** is a string instead of a function, groups by the property named by **iterator** on each of the values.

```
view _.groupBy [1.3, 2.1, 2.4], (num) -> Math.floor num
```

```
view _.groupBy ['one', 'two', 'three'], 'length'
```

sortedIndex `_.sortedIndex list, value, [iterator]`

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 **iterator** is passed, it will be used to compute the sort ranking of each value.

```
show _.sortedIndex [10, 20, 30, 40, 50], 35
```

shuffle `_.shuffle list`

Returns a shuffled copy of the **list**, using a version of the Fisher-Yates shuffle.

```
show _.shuffle [1..6]
```

toArray `_.toArray list`

Converts the **list** (anything that can be iterated over), into a real Array. Useful for transmuting the **arguments** object.

```
(-> show _.toArray(arguments).slice(0))(1, 2, 3)
```

size `_.size list`

Return the number of values in the **list**.

```
show _.size {one : 1, two : 2, three : 3}
```

Array Functions

*Note: All array functions will also work on the **arguments** object.*

first `_.first array, [n]` Alias: **head**

Returns the first element of an **array**. Passing **n** will return the first **n** elements of the array.

```
show _.first [5, 4, 3, 2, 1]
```

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.

```
show _.initial [5, 4, 3, 2, 1]
```

last `_.last array, [n]`

Returns the last element of an **array**. Passing **n** will return the last **n** elements of the array.

```
show _.last [5, 4, 3, 2, 1]
```

rest `_.rest array, [index]` Alias: **tail**

Returns the **rest** of the elements in an array. Pass an **index** to return the values of the array from that index onward.

```
show _.rest [5, 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.

```
show _.compact [0, 1, false, 2, '', 3]
```

flatten `_.flatten array`

Flattens a nested **array** (the nesting can be to any depth).

```
show _.flatten [1, [2], [3, [[[4]]]]]
```

without `_.without array, [*values]`

Returns a copy of the **array** with all instances of the **values** removed. === is used for the equality test.

```
show _.without [1, 2, 1, 0, 3, 1, 4], 0, 1
```

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**.

```
show _.union [1, 2, 3], [101, 2, 1, 10], [2, 1]
```

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**.

```
show _.intersection [1, 2, 3], [101, 2, 1, 10], [2, 1]
```

difference `_.difference array, *others`

Similar to **without**, but returns the values from **array** that are not present in the **other** arrays.

```
show _.difference [1, 2, 3, 4, 5], [5, 2, 10]
```

uniq `_.uniq array, [isSorted], [iterator]` Alias: **unique**

Produces a duplicate-free version of the **array**, using === to test object equality. 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 **iterator** function.

```
show _.uniq [1, 2, 1, 3, 1, 4]
```

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.

```
show _.zip ['moe', 'larry', 'curly'], [30, 40, 50], [true, false, false]
```

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**. Uses the native **indexOf** function unless it's missing. 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.

```
show _.indexOf [1, 2, 3], 2
```

lastIndexOf `_.lastIndexOf array, value`

Returns the index of the last occurrence of **value** in the **array**, or *-1* if value is not present. Uses the native **lastIndexOf** function if possible.

```
show _.lastIndexOf [1, 2, 3, 1, 2, 3], 2
```

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** to **stop**, incremented (or decremented) by **step**, exclusive.

```
show _.range 10
show _.range 1, 11
show _.range 0, 30, 5
show _.range 0, -10, -1
show _.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, bind **arguments** to the **function** to pre-fill them, also known as **currying**.

```
func = (greeting) -> greeting + ': ' + this.name
func = _.bind func, {name : 'moe'}, 'hi'
show func()
```

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*. If no **methodNames** are provided, all of the object's function properties will be bound to it.

```
buttonView = {
  label   : 'underscore'
  onClick : -> show 'clicked: ' + this.label
  onHover : -> show 'hovering: ' + this.label
}
_.bindAll buttonView
jQuery('#underscore_button').bind 'click', buttonView.onClick
```

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.

```
timeIt = (func, a...) ->
  before = new Date
  result = func a...
  show "Elapsed: #{new Date - before}ms"
  result

fibonacci = _.memoize (n) ->
  if n < 2 then n else fibonacci(n - 1) + fibonacci(n - 2)

show timeIt fibonacci, 1000
show timeIt fibonacci, 1000
```

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.

```
log = _.bind show, console
_.delay log, 1, 'logged later'
# See the end of this document for the output
```

defer `_.defer function`

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.

```
_.defer -> show 'deferred'
# See the end of this document for the output
```

throttle `_.throttle function, wait`

Returns a throttled version of the function, that, when invoked repeatedly, will only actually call the wrapped function at most once per every **wait** milliseconds. Useful for rate-limiting events that occur faster than you can keep up with.

```
updatePosition = (evt) -> show "Position #{evt}"
throttled = _.throttle updatePosition, 100
for i in [0..10]
  throttled i
# $(window).scroll throttled
```

debounce `_.debounce function, wait`

Calling a debounced function will postpone its execution until after **wait** milliseconds have elapsed since the last time the function 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…

```
calculateLayout = -> show "It's quiet now"
lazyLayout = _.debounce calculateLayout, 100
lazyLayout()
# $(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.

```
createApplication = -> show "Created"
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.

```
skipFirst = _.after 3, show
for i in [0..3]
  skipFirst i
```

```
# renderNotes is run once, after all notes have saved.
renderNotes = _.after notes.length, render
_.each notes, (note) ->
  note.asyncSave {success: renderNotes}
```

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.

```
hello = (name) -> "hello: " + name
hello = _.wrap hello, (func) ->
  "before, #{func "moe"}, after"
show hello()
```

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()))*.

```
greet    = (name) -> "hi: " + name
exclaim  = (statement) -> statement + "!"
welcome = _.compose exclaim, greet
show welcome 'moe'
```

Object Functions

keys `_.keys object`

Retrieve all the names of the **object**'s properties.

```
show _.keys {one : 1, two : 2, three : 3}
```

values `_.values object`

Return all of the values of the **object**'s properties.

```
show _.values {one : 1, two : 2, three : 3}
```

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.

```
show _.functions _
```

extend `_.extend destination, *sources`

Copy all of the properties in the **source** objects over to the **destination** object. It's in-order, so the last source will override properties of the same name in previous arguments.

```
view _.extend {name : 'moe'}, {age : 50}
```

defaults `_.defaults object, *defaults`

Fill in missing properties in **object** with default values from the **defaults** objects. As soon as the property is filled, further defaults will have no effect.

```
iceCream = {flavor : "chocolate"}
view _.defaults iceCream, {flavor : "vanilla", sprinkles : "lots"}
```

clone `_.clone object`

Create a shallow-copied clone of the **object**. Any nested objects or arrays will be copied by reference, not duplicated.

```
view _.clone {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.

```
show _([1,2,3,200]).chain().
  filter((num) -> num % 2 is 0).
  tap(show).
  map((num) -> num * num).
  value()
```

isEqual `_.isEqual object, other`

Performs an optimized deep comparison between the two objects, to determine if they should be considered equal.

```
moe   = {name : 'moe', luckyNumbers : [13, 27, 34]}
clone = {name : 'moe', luckyNumbers : [13, 27, 34]}
moe is clone
show _.isEqual(moe, clone)
```

isEmpty `_.isEmpty object`

Returns *true* if **object** contains no values.

```
show _.isEmpty([1, 2, 3])
show _.isEmpty({})
```

isElement `_.isElement object`

Returns *true* if **object** is a DOM element.

```
show _.isElement document?.getElementById 'page'
```

isArray `_.isArray object`

Returns *true* if **object** is an Array.

```
show (-> _.isArray arguments)()
show _.isArray [1,2,3]
```

isArguments `_.isArguments object`

Returns *true* if **object** is an Arguments object.

```
show (-> _.isArguments arguments)(1, 2, 3)
show _.isArguments [1,2,3]
```

isFunction `_.isFunction object`

Returns *true* if **object** is a Function.

```
show _.isFunction console.debug
```

isString `_.isString object`

Returns *true* if **object** is a String.

```
show _.isString "moe"
```

isNumber `_.isNumber object`

Returns *true* if **object** is a Number.

```
show _.isNumber 8.4 * 5
```

isBoolean `_.isBoolean object`

Returns *true* if **object** is either *true* or *false*.

```
show _.isBoolean null
```

isDate `_.isDate object`

Returns *true* if **object** is a Date.

```
show _.isDate new Date()
```

isRegExp `_.isRegExp object`

Returns *true* if **object** is a RegExp.

```
show _.isRegExp /moe/
```

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 if the variable is *undefined*.

```
show _.isNaN NaN
show isNaN undefined
show _.isNaN undefined
```

isNull `_.isNull object`

Returns *true* if the value of **object** is *null*.

```
show _.isNull null
show _.isNull undefined
```

isUndefined `_.isUndefined variable`

Returns *true* if **variable** is *undefined*.

```
show _.isUndefined window?.missingVariable
```

Utility Functions

noConflict `_.noConflict`

Give control of the "_" variable back to its previous owner. Returns a reference to the **Underscore** object.

```
# The examples will stop working if this is enabled
# 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 iterator.

```
moe = {name : 'moe'}
show moe is _.identity(moe)
```

times `_.times n, iterator`

Invokes the given iterator function **n** times.

```
(genie = {}).grantWish = -> show 'Served'
_(3).times -> genie.grantWish()
```

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 : (string) ->
    string.charAt(0).toUpperCase() +
    string.substring(1).toLowerCase()
show _("fabio").capitalize()
```

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.

```
show _.uniqueId 'contact_'
show _.uniqueId 'contact_'
```

escape `_.escape string`

Escapes a string for insertion into HTML, replacing `&`, `<`, `>`, `"`, `'`, and `/` characters.

```
show _.escape 'Curly, Larry & Moe'
```

template `_.template templateString, [context]`

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 variables, 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 **context** object that has properties corresponding to the template's free variables. If you're writing a one-off, you can pass the **context** object as the second parameter to **template** in order to render immediately instead of returning a template function.

```
compiled = _.template "hello: <%= name %>"
show compiled name : 'moe'
```

```
list = "<% _.each(people, function(name) { %> <li><%= name %></li> <% }); %>"
show _.escape _.template list, people : ['moe', 'curly', 'larry']
```

```
template = _.template "<b><%- value %></b>"
show _.escape template value : '<script>'
```

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

```
compiled = _.template "<% print('Hello ' + epithet) %>"
show compiled {epithet: "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, and an (optional) **evaluate** regex to match expressions that should be inserted and evaluated, respectively. If no **evaluate** regex is provided, your templates will only be capable of interpolating values. For example, to perform Mustache.js style templating:

```
saveSettings = _.templateSettings
_.templateSettings = interpolate : /\{\{(.+?)\}\}/g

template = _.template "Hello {{ name }}!"
show template name : "Mustache"

_.templateSettings = saveSettings
```

Chaining

chain `_(obj).chain`

Returns a wrapped object. Calling methods on this object will continue to return wrapped objects until `value` is used. (A more realistic example.)

```
stooges = [
  {name : 'curly', age : 25}
  {name : 'moe', age : 21}
  {name : 'larry', age : 23}
]
youngest = _(stooges).chain()
  .sortBy((stooge) -> stooge.age)
  .map((stooge) -> stooge.name + ' is ' + stooge.age)
  .first()
  .value()
show youngest
```

value `_(obj).value`

Extracts the value of a wrapped object.

```
show _([1, 2, 3]).value()
```

The end

```
show 'Delayed output will show up here'
```

Output

```
[ 2, 4, 6 ]
[ 2, 4, 6 ]
{
  I'm: 2,
  a: 2,
  lumberjack: 2,
  and: 4,
  okay: 2,
  I: 2,
  sleep: 1,
  all: 4,
  night: 2,
  work: 1,
  day: 2,
  He's: 1,
  he's: 1,
  He: 1,
  sleeps: 1,
  he: 1,
  works: 1
}
1
2
3
1
2
3
[ 3, 6, 9 ]
[ 3, 6, 9 ]
6
[ 4, 5, 2, 3, 0, 1 ]
2
[ 2, 4, 6 ]
[ 1, 3, 5 ]
false
true
true
[ [ 1, 5, 7 ], [ 1, 2, 3 ] ]
[ 'moe', 'larry', 'curly' ]
{
  name: curly,
  age: 60
}
2
[ 5, 4, 6, 3, 1, 2 ]
{
  1: 1.3,
  2: 2.1,2.4
}
{
  3: one,two,
  5: three
}
3
[ 4, 3, 2, 5, 6, 1 ]
[ 1, 2, 3 ]
3
5
[ 5, 4, 3, 2 ]
1
[ 4, 3, 2, 1 ]
[ 1, 2, 3 ]
[ 1, 2, 3, 4 ]
[ 2, 3, 4 ]
[ 1, 2, 3, 101, 10 ]
[ 1, 2 ]
[ 1, 3, 4 ]
[ 1, 2, 3, 4 ]
[ [ 'moe', 30, true ],
  [ 'larry', 40, false ],
  [ 'curly', 50, false ] ]
1
4
[ 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 ]
[ 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 ]
[ 0, 5, 10, 15, 20, 25 ]
[ 0, -1, -2, -3, -4, -5, -6, -7, -8, -9 ]
[]
hi: moe
Elapsed: 2ms
4.346655768693743e+208
Elapsed: 0ms
4.346655768693743e+208
Position 0
Created
2
3
before, hello: moe, after
hi: moe!
[ 'one', 'two', 'three' ]
[ 1, 2, 3 ]
[ '_',
  'after',
  'all',
  'any',
  'bind',
  'bindAll',
  'clone',
  'compact',
  'compose',
  'contains',
  'debounce',
  'defaults',
  'defer',
  'delay',
  'detect',
  'difference',
  'each',
  'escape',
  'every',
  'extend',
  'filter',
  'find',
  'first',
  'flatten',
  'foldl',
  'foldr',
  'forEach',
  'functions',
  'groupBy',
  'head',
  'identity',
  'include',
  'indexOf',
  'initial',
  'inject',
  'intersect',
  'intersection',
  'invoke',
  'isArguments',
  'isArray',
  'isBoolean',
  'isDate',
  'isElement',
  'isEmpty',
  'isEqual',
  'isFunction',
  'isNaN',
  'isNull',
  'isNumber',
  'isObject',
  'isRegExp',
  'isString',
  'isUndefined',
  'keys',
  'last',
  'lastIndexOf',
  'map',
  'max',
  'memoize',
  'methods',
  'min',
  'mixin',
  'noConflict',
  'once',
  'pluck',
  'range',
  'reduce',
  'reduceRight',
  'reject',
  'rest',
  'select',
  'shuffle',
  'size',
  'some',
  'sortBy',
  'sortedIndex',
  'tail',
  'tap',
  'template',
  'throttle',
  'times',
  'toArray',
  'union',
  'uniq',
  'unique',
  'uniqueId',
  'values',
  'without',
  'wrap',
  'zip' ]
{
  name: moe,
  age: 50
}
{
  flavor: chocolate,
  sprinkles: lots
}
{
  name: moe
}
[ 2, 200 ]
[ 4, 40000 ]
true
false
true
false
false
```

```
true
true
false
false
true
true
false
true
true
true
true
false
true
false
true
true
Served
Served
Served
Fabio
contact_0
contact_1
Curly, Larry &amp; Moe
hello: moe
 &lt;li&gt;moe&lt;&#x2F;li&gt;  &lt;li&gt;curly&lt;&#x2F;li&gt;  &lt;li&gt;larry&lt;&#x2F;li&gt;
&lt;b&gt;&amp;lt;script&amp;gt;&lt;&#x2F;b&gt;
Hello stooge
Hello Mustache!
moe is 21
[ 1, 2, 3 ]
Delayed output will show up here
logged later
deferred
Position 10
It’s quiet now
```

JavaScript

```
(function() {
  var calculateLayout, clone, compiled, createApplication, even, evens, exclaim, fibonacci, flat, func, genie, greet, hello, i, iceC
    __hasProp = Object.prototype.hasOwnProperty,
    __slice = Array.prototype.slice;

  show = console.log;

  showDocument = function(doc, width, height) {
    return show(doc);
  };

  if (typeof exports !== ”undefined” && exports !== null) {
    _ = require(’underscore’);
  } else {
    _ = window._;
  }

  view = function(obj) {
    var k, v;
    return show(typeof obj === ’object’ ? ”{” + ((function() {
      var _results;
      _results = [];
      for (k in obj) {
        if (!__hasProp.call(obj, k)) continue;
        v = obj[k];
        _results.push(”\n  ” + k + ”: ” + v);
      }
      return _results;
    })()) + ”\n}” : obj);
  };

  tryIt = function() {
    show(view);
    return view({
      'JavaScript': "we could have been the closest of friends",
      'EcmaScript': "we might have been the world's greatest lovers",
      'But': "now we're just without each other"
    });
  };

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

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

  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"
    }
  ];

  view(_(lyrics).chain().map(function(line) {
    return line.words.split(" ");
  }).flatten().reduce((function(counts, word) {
    counts[word] = (counts[word] || 0) + 1;
    return counts;
  }), {}).value());

  _.each([1, 2, 3], function(num) {
    return show(num);
  });

  _.each({
    one: 1,
    two: 2,
    three: 3
  }, function(num, key) {
    return show(num);
  });

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

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

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

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

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

```

show(flat);

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

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

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

show(_.all([true, 1, null, 'yes'], _.identity));

show(_.any([null, 0, 'yes', false]));

show(_.include([1, 2, 3], 3));

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

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

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

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

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

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

show(_.min(numbers));

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

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

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

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

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

```

(function() {
  return show(_.toArray(arguments).slice(0));
})(1, 2, 3);

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

show(_.range(10));

show(_.range(1, 11));

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

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

show(_.range(0));

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

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

show(func());

timeIt = function() {
  var a, before, func, result;
  func = arguments[0], a = 2 <= arguments.length ? __slice.call(arguments, 1) : [];
  before = new Date;
  result = func.apply(null, a);
  show("Elapsed: " + (new Date - before) + "ms");
  return result;
};

fibonacci = _.memoize(function(n) {
  if (n < 2) {
    return n;
  } else {
    return fibonacci(n - 1) + fibonacci(n - 2);
  }
});

show(timeIt(fibonacci, 1000));

show(timeIt(fibonacci, 1000));

log = _.bind(show, console);

_.delay(log, 1, ’logged later’);

_.defer(function() {
  return show(’deferred’);
});

updatePosition = function(evt) {
  return show(”Position ” + evt);
};

throttled = _.throttle(updatePosition, 100);

for (i = 0; i <= 10; i++) {
  throttled(i);
}

calculateLayout = function() {
  return show(”It’s quiet now”);
};

lazyLayout = _.debounce(calculateLayout, 100);

lazyLayout();

createApplication = function() {
  return show(”Created”);
};

initialize = _.once(createApplication);

initialize();

initialize();

skipFirst = _.after(3, show);

for (i = 0; i <= 3; i++) {
  skipFirst(i);
}

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

hello = _.wrap(hello, function(func) {
  return ”before, ” + (func(”moe”)) + ”, after”;
});

show(hello());

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

exclaim = function(statement) {
  return statement + ”!”;
};

welcome = _.compose(exclaim, greet);

show(welcome(’moe’));

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

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

show(_.functions(_));

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

iceCream = {
  flavor: "chocolate"
};

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

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

show(_([1, 2, 3, 200]).chain().filter(function(num) {
  return num % 2 === 0;
}).tap(show).map(function(num) {
  return num * num;
}).value());

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

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

moe === clone;

show(_.isEqual(moe, clone));

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

show(_.isEmpty({}));

show(_.isElement(typeof document !== "undefined" && document !== null ? document.getElementById('page') : void 0));

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

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

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

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

show(_.isFunction(console.debug));
show(_.isString("moe"));

show(_.isNumber(8.4 * 5));

show(_.isBoolean(null));

show(_.isDate(new Date()));

show(_.isRegExp(/moe/));

show(_.isNaN(NaN));

show(isNaN(void 0));

show(_.isNaN(void 0));

show(_.isNull(null));

show(_.isNull(void 0));

show(_.isUndefined(typeof window !== "undefined" && window !== null ? window.missingVariable : void 0));

moe = {
  name: 'moe'
};

show(moe === _.identity(moe));

(genie = {}).grantWish = function() {
  return show('Served');
};

_(3).times(function() {
  return genie.grantWish();
});

_.mixin({
  capitalize: function(string) {
    return string.charAt(0).toUpperCase() + string.substring(1).toLowerCase();
  }
});

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

show(_.uniqueId('contact_'));

show(_.uniqueId('contact_'));

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

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

show(compiled({
  name: 'moe'
}));

list = "<% _.each(people, function(name) { %> <li><%= name %></li> <% }); %>";

show(_.escape(_.template(list, {
  people: ['moe', 'curly', 'larry']
})));

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

show(_.escape(template({
  value: '<script>'
})));

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

show(compiled({
    epithet: ”stooge”
  }));

  saveSettings = _.templateSettings;

  _.templateSettings = {
    interpolate: /\{\{(.+?)\}\}/g
  };

  template = _.template(”Hello {{ name }}!”);

  show(template({
    name: ”Mustache”
  }));

  _.templateSettings = saveSettings;

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

  youngest = _(stooges).chain().sortBy(function(stooge) {
    return stooge.age;
  }).map(function(stooge) {
    return stooge.name + ’ is ’ + stooge.age;
  }).first().value();

  show(youngest);

  show(_([1, 2, 3]).value());

  show(’Delayed output will show up here’);

}).call(this);
```

Formats CoffeeScript Markdown PDF HTML