[**Underscore.php**](http://github.com/brianhaveri/Underscore.php)**by**[**brianhaveri**](http://github.com/brianhaveri)

[Underscore.php](http://github.com/brianhaveri/Underscore.php) is a PHP port of the popular [Underscore.js](http://documentcloud.github.com/underscore/) library. In addition to porting Underscore's functionality, Underscore.php includes [matching unit tests](https://github.com/brianhaveri/Underscore.php/tree/master/test). Underscore.php requires PHP 5.3 or greater.

Underscore.php is [hosted on GitHub](http://github.com/brianhaveri/Underscore.php) and open sourced under the MIT license.

**Download**

You can download this project in either [zip](http://github.com/brianhaveri/Underscore.php/zipball/master) or [tar](http://github.com/brianhaveri/Underscore.php/tarball/master) formats.

**Table of Contents**

[Object-Oriented and Static Styles](https://brianhaveri.github.io/Underscore.php/#styles)

**Collections**

[each](https://brianhaveri.github.io/Underscore.php/#each), [map](https://brianhaveri.github.io/Underscore.php/#map), [reduce](https://brianhaveri.github.io/Underscore.php/#reduce), [reduceRight](https://brianhaveri.github.io/Underscore.php/#reduceRight), [find](https://brianhaveri.github.io/Underscore.php/#find), [filter](https://brianhaveri.github.io/Underscore.php/#filter), [reject](https://brianhaveri.github.io/Underscore.php/#reject), [all](https://brianhaveri.github.io/Underscore.php/#all), [any](https://brianhaveri.github.io/Underscore.php/#any), [includ](https://brianhaveri.github.io/Underscore.php/#includ), [invoke](https://brianhaveri.github.io/Underscore.php/#invoke), [pluck](https://brianhaveri.github.io/Underscore.php/#pluck), [max](https://brianhaveri.github.io/Underscore.php/#max), [min](https://brianhaveri.github.io/Underscore.php/#min), [groupBy](https://brianhaveri.github.io/Underscore.php/#groupBy),[sortBy](https://brianhaveri.github.io/Underscore.php/#sortBy), [sortedIndex](https://brianhaveri.github.io/Underscore.php/#sortedIndex), [shuffle](https://brianhaveri.github.io/Underscore.php/#shuffle), [toArray](https://brianhaveri.github.io/Underscore.php/#toArray), [size](https://brianhaveri.github.io/Underscore.php/#size)

**Arrays**

[first](https://brianhaveri.github.io/Underscore.php/#first), [initial](https://brianhaveri.github.io/Underscore.php/#initial), [rest](https://brianhaveri.github.io/Underscore.php/#rest), [last](https://brianhaveri.github.io/Underscore.php/#last), [compact](https://brianhaveri.github.io/Underscore.php/#compact), [flatten](https://brianhaveri.github.io/Underscore.php/#flatten), [without](https://brianhaveri.github.io/Underscore.php/#without), [uniq](https://brianhaveri.github.io/Underscore.php/#uniq), [union](https://brianhaveri.github.io/Underscore.php/#union), [intersection](https://brianhaveri.github.io/Underscore.php/#intersection), [difference](https://brianhaveri.github.io/Underscore.php/#difference), [zip](https://brianhaveri.github.io/Underscore.php/#zip), [indexOf](https://brianhaveri.github.io/Underscore.php/#indexOf),[lastIndexOf](https://brianhaveri.github.io/Underscore.php/#lastIndexOf), [range](https://brianhaveri.github.io/Underscore.php/#range)

**Functions**

bind, bindAll [memoize](https://brianhaveri.github.io/Underscore.php/#memoize), delay, defer, [throttle](https://brianhaveri.github.io/Underscore.php/#throttle), debounce, [once](https://brianhaveri.github.io/Underscore.php/#once), [after](https://brianhaveri.github.io/Underscore.php/#after), [wrap](https://brianhaveri.github.io/Underscore.php/#wrap), [compose](https://brianhaveri.github.io/Underscore.php/#compose)

**Objects**

[keys](https://brianhaveri.github.io/Underscore.php/#keys), [values](https://brianhaveri.github.io/Underscore.php/#values), [functions](https://brianhaveri.github.io/Underscore.php/#functions), [extend](https://brianhaveri.github.io/Underscore.php/#extend), [defaults](https://brianhaveri.github.io/Underscore.php/#defaults), [clon](https://brianhaveri.github.io/Underscore.php/#clon), [tap](https://brianhaveri.github.io/Underscore.php/#tap), [has](https://brianhaveri.github.io/Underscore.php/#has), [isEqual](https://brianhaveri.github.io/Underscore.php/#isEqual), [isEmpty](https://brianhaveri.github.io/Underscore.php/#isEmpty), isElement, [isObject](https://brianhaveri.github.io/Underscore.php/#isObject), [isArray](https://brianhaveri.github.io/Underscore.php/#isArray),isArguments, [isFunction](https://brianhaveri.github.io/Underscore.php/#isFunction), [isString](https://brianhaveri.github.io/Underscore.php/#isString), [isNumber](https://brianhaveri.github.io/Underscore.php/#isNumber), [isBoolean](https://brianhaveri.github.io/Underscore.php/#isBoolean), [isDate](https://brianhaveri.github.io/Underscore.php/#isDate), isRegExp, [isNaN](https://brianhaveri.github.io/Underscore.php/#isNaN), [isNull](https://brianhaveri.github.io/Underscore.php/#isNull), isUndefined

**Utility**

noConflict, [identity](https://brianhaveri.github.io/Underscore.php/#identity), [times](https://brianhaveri.github.io/Underscore.php/#times), [mixin](https://brianhaveri.github.io/Underscore.php/#mixin), [uniqueId](https://brianhaveri.github.io/Underscore.php/#uniqueId), [escape](https://brianhaveri.github.io/Underscore.php/#escape), [template](https://brianhaveri.github.io/Underscore.php/#template)

**Chaining**

[chain](https://brianhaveri.github.io/Underscore.php/#chain), [value](https://brianhaveri.github.io/Underscore.php/#value)

*Some functions were not ported from Underscore.js to Underscore.php for technical reasons or because they weren't applicable to PHP.*

**Single Underscore vs. Double Underscore**

In many PHP installations, \_() already exists as an alias to gettext(). The previous declaration of \_ in PHP forces Underscore.php to use another name. For consistency and memorability, \_\_ has been chosen for both the function and class name.

**Object-Oriented and Static Styles**

Underscore.php works in both object-oriented and static styles. The following lines are identical ways to double a list of numbers.

\_\_::map(array(1, 2, 3), function($n) { return $n \* 2; });

\_\_(array(1, 2, 3))->map(function($n) { return $n \* 2; });

**Collections (Arrays or Objects)**

**each** \_\_::each(collection, iterator)

Iterates over the collection and yield each in turn to the iterator function. Arguments passed to iterator are (value, key, collection). Unlike Underscore.js, context is passed using PHP's use statement. Underscore.php does not contain the forEach alias because 'foreach' is a reserved keyword in PHP.

\_\_::each(array(1, 2, 3), function($num) { echo $num . ','; }); // 1,2,3,

$multiplier = 2;

\_\_::each(array(1, 2, 3), function($num, $index) use ($multiplier) {

echo $index . '=' . ($num \* $multiplier) . ',';

});

// 0=2,1=4,2=6,

**map** \_\_::map(collection, iterator)*Alias:****collect***

Returns an array of values by mapping each in collection through the iterator. Arguments passed to iterator are (value, key, collection). Unlike Underscore.js, context is passed using PHP's use statement.

\_\_::map(array(1, 2, 3), function($num) { return $num \* 3; }); // array(3, 6, 9)

\_\_::map(array('one'=>1, 'two'=>2, 'three'=>3), function($num, $key) {

return $num \* 3;

});

// array(3, 6, 9);

**reduce** \_\_::reduce(collection, iterator, memo)*Aliases:****inject, foldl***

Reduce the collection into a single value. Memo is the initial state of the reduction, updated by the return value of the iterator. Unlike Underscore.js, context is passed using PHP's use statement.

\_\_::reduce(array(1, 2, 3), function($memo, $num) { return $memo + $num; }, 0); // 6

**reduceRight** \_\_::reduceRight(collection, iterator, memo)*Alias:****foldr***

Right-associative version of reduce.

$list = array(array(0, 1), array(2, 3), array(4, 5));

$flat = \_\_::reduceRight($list, function($a, $b) { return array\_merge($a, $b); }, array());

// array(4, 5, 2, 3, 0, 1)

**find** \_\_::find(collection, iterator)*Alias:****detect***

Return the value of the first item in the collection that passes the truth test (**iterator**).

\_\_::find(array(1, 2, 3, 4), function($num) { return $num % 2 === 0; }); // 2

**filter** \_\_::filter(collection, iterator)*Alias:****select***

Return the values in the collection that pass the truth test (**iterator**).

\_\_::filter(array(1, 2, 3, 4), function($num) { return $num % 2 === 0; }); // array(2, 4)

**reject** \_\_::reject(collection, iterator)

Return an array where the items failing the truth test (**iterator**) are removed.

\_\_::reject(array(1, 2, 3, 4), function($num) { return $num % 2 === 0; }); // array(1, 3)

**all** \_\_::all(collection, iterator)

Returns true if all values in the collection pass the truth test (**iterator**).

\_\_::all(array(1, 2, 3, 4), function($num) { return $num % 2 === 0; }); // false

\_\_::all(array(1, 2, 3, 4), function($num) { return $num < 5; }); // true

**any** \_\_::any(collection, iterator)

Returns true if any values in the collection pass the truth test (**iterator**).

\_\_::any(array(1, 2, 3, 4), function($num) { return $num % 2 === 0; }); // true

\_\_::any(array(1, 2, 3, 4), function($num) { return $num === 5; }); // false

**includ** \_\_::includ(collection, value)*Alias:****contains***

Returns true if value is found in the collection using === to test equality. This function is called 'include' in Underscore.js, but was renamed to 'includ' in Underscore.php because 'include' is a reserved keyword in PHP.

\_\_::includ(array(1, 2, 3), 3); // true

**invoke** \_\_::invoke(collection, functionName)

Returns a copy of the collection after running functionName across all elements.

\_\_::invoke(array(' foo', ' bar '), 'trim'); // array('foo', 'bar')

**pluck** \_\_::pluck(collection, propertyName)

Extract an array of property values

$stooges = array(

array('name'=>'moe', 'age'=>40),

array('name'=>'larry', 'age'=>50),

array('name'=>'curly', 'age'=>60)

);

\_\_::pluck($stooges, 'name'); // array('moe', 'larry', 'curly')

**max** \_\_::max(collection, [iterator])

Returns the maximum value from the collection. If passed an iterator, max will return max value returned by the iterator. Unlike Underscore.js, context is passed using PHP's use statement.

$stooges = array(

array('name'=>'moe', 'age'=>40),

array('name'=>'larry', 'age'=>50),

array('name'=>'curly', 'age'=>60)

);

\_\_::max($stooges, function($stooge) { return $stooge['age']; });

// array('name'=>'curly', 'age'=>60)

**min** \_\_::min(collection, [iterator])

Returns the minimum value from the collection. If passed an iterator, min will return min value returned by the iterator. Unlike Underscore.js, context is passed using PHP's use statement.

$stooges = array(

array('name'=>'moe', 'age'=>40),

array('name'=>'larry', 'age'=>50),

array('name'=>'curly', 'age'=>60)

);

\_\_::min($stooges, function($stooge) { return $stooge['age']; });

// array('name'=>'moe', 'age'=>40)

**groupBy** \_\_::groupBy(collection, iterator)

Group values by their return value when passed through the iterator. If iterator is a string, the result will be grouped by that property.

\_\_::groupBy(array(1, 2, 3, 4, 5), function($n) { return $n % 2; });

// array(0=>array(2, 4), 1=>array(1, 3, 5))

$values = array(

array('name'=>'Apple', 'grp'=>'a'),

array('name'=>'Bacon', 'grp'=>'b'),

array('name'=>'Avocado', 'grp'=>'a')

);

\_\_::groupBy($values, 'grp');

//array(

// 'a'=>array(

// array('name'=>'Apple', 'grp'=>'a'),

// array('name'=>'Avocado', 'grp'=>'a')

// ),

// 'b'=>array(

// array('name'=>'Bacon', 'grp'=>'b')

// )

//);

**sortBy** \_\_::sortBy(collection, iterator)

Returns an array sorted in ascending order based on the iterator results. If passed an iterator, min will return min value returned by the iterator. Unlike Underscore.js, context is passed using PHP's use statement.

\_\_::sortBy(array(1, 2, 3), function($n) { return -$n; }); // array(3, 2, 1)

**sortedIndex** \_\_::sortedIndex(collection, value, [iterator])

Returns the index at which the value should be inserted into the sorted collection.

\_\_::sortedIndex(array(10, 20, 30, 40), 35); // 3

**shuffle** \_\_::shuffle(collection)

Returns a shuffled copy of the collection.

\_\_::shuffle(array(10, 20, 30, 40)); // 30, 20, 40, 10

**toArray** \_\_::toArray(collection)

Converts the collection into an array.

$stooge = new StdClass;

$stooge->name = 'moe';

$stooge->age = 40;

\_\_::toArray($stooge); // array('name'=>'moe', 'age'=>40)

**size** \_\_::size(collection)

Returns the number of values in the collection.

$stooge = new StdClass;

$stooge->name = 'moe';

$stooge->age = 40;

\_\_::size($stooge); // 2

**Arrays**

**first** \_\_::first(array, [n])*Alias:****head***

Get the first element of an array. Passing n returns the first n elements.

\_\_::first(array(5, 4, 3, 2, 1)); // 5

\_\_::first(array(5, 4, 3, 2, 1), 3); // array(5, 4, 3)

**initial** \_\_::initial(array, [n])

Get everything but the last array element. Passing n excludes the last n elements.

\_\_::initial(array(5, 4, 3, 2, 1)); // array(5, 4, 3, 2)

\_\_::initial(array(5, 4, 3, 2, 1), 3); // array(5, 4)

**rest** \_\_::rest(array, [index])*Alias:****tail***

Get the rest of the array elements. Passing an index returns from that index onward.

\_\_::rest(array(5, 4, 3, 2, 1)); // array(4, 3, 2, 1)

**last** \_\_::last(array, [n])

Get the last element of an array. Passing n returns the last n elements.

\_\_::last(array(5, 4, 3, 2, 1)); // 1

\_\_::last(array(5, 4, 3, 2, 1), 2); // array(2, 1)

**compact** \_\_::compact(array)

Returns a copy of the array with falsy values removed

\_\_::compact(array(false, true, 'a', 0, 1, '')); // array(true, 'a', 1)

**flatten** \_\_::flatten(array, [shallow])

Flattens a multidimensional array. If you pass shallow, the array will only be flattened a single level.

\_\_::flatten(array(1, array(2), array(3, array(array(array(4))))));

// array(1, 2, 3, 4)

\_\_::flatten(array(1, array(2), array(3, array(array(array(4))))), true);

// array(1, 2, 3, array(array(4)))

**without** \_\_::without(array, [\*values])

Returns a copy of the array with all instances of **values** removed. === is used for equality testing. Keys are maintained.

\_\_::without(array(5, 4, 3, 2, 1), 3, 2); // array(5, 4, 4=>1)

**uniq** \_\_::uniq(array, [isSorted [, iterator]])*Alias:****unique***

Returns a copy of the array containing no duplicate values. Unlike Underscore.js, passing isSorted does not currently affect the performance of uniq. You can optionally compute uniqueness by passing an iterator function.

\_\_::uniq(array(2, 2, 4, 4, 4, 1, 1, 1)); // array(2, 4, 1)

**union** \_\_::union(\*arrays)

Returns an array containing the unique items in one or more of the arrays.

$arr1 = array(1, 2, 3);

$arr2 = array(101, 2, 1, 10);

$arr3 = array(2, 1);

\_\_::union($arr1, $arr2, $arr3); // array(1, 2, 3, 101, 10)

**intersection** \_\_::intersection(\*arrays)

Returns an array containing the intersection of all the arrays. Each value in the resulting array exists in all arrays.

$arr1 = array(0, 1, 2, 3);

$arr2 = array(1, 2, 3, 4);

$arr3 = array(2, 3, 4, 5);

\_\_::intersection($arr1, $arr2, $arr3); // array(2, 3)

**difference** \_\_::difference(array, \*others)

Returns an array containing the items existing in one array, but not the other.

\_\_::difference(array(1, 2, 3, 4, 5), array(5, 2, 10)); // array(1, 3, 4)

**zip** \_\_::zip(\*arrays)

Merges arrays

$names = array('moe', 'larry', 'curly');

$ages = array(30, 40, 50);

$leaders = array(true, false, false);

\_\_::zip($names, $ages, $leaders);

// array(

// array('moe', 30, true),

// array('larry', 40, false),

// array('curly', 50, false)

// )

**indexOf** \_\_::indexOf(array, value)

Returns the index of the first match. Returns -1 if no match is found. Unlike Underscore.js, Underscore.php does not take a second isSorted parameter.

\_\_::indexOf(array(1, 2, 3, 2, 2), 2); // 1

**lastIndexOf** \_\_::lastIndexOf(array, value)

Returns the index of the last match. Returns -1 if no match is found.

\_\_::lastIndexOf(array(1, 2, 3, 2, 2), 2); // 4

**range** \_\_::range([start], stop, [step])

Returns an array of integers from **start** to **stop** (exclusive) by **step**. Defaults: **start**=0, **step**=1.

\_\_::range(10); // array(0, 1, 2, 3, 4, 5, 6, 7, 8, 9)

\_\_::range(1, 5); // array(1, 2, 3, 4)

\_\_::range(0, 30, 5); // array(0, 5, 10, 15, 20, 25)

\_\_::range(0, -5, -1); // array(0, -1, -2, -3, -4)

\_\_::range(0); // array()

**Functions**

**memoize** \_\_::memoize(function, [hashFunction])

Memoizes a function by caching the computed result. Useful for computationally expensive functions. Optionally, pass a hashFunction to calculate the key for the cached value.

$fibonacci = function($n) use (&$fibonacci) {

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

};

$fastFibonacci = \_\_::memoize($fibonacci);

**throttle** \_\_::throttle(function, wait)

Throttles a function so that it can only be called once every **wait** milliseconds.

$func = function() { return 'x'; }

\_\_::throttle($func);

**once** \_\_::once(function)

Creates a version of the function that can only be called once.

$num = 0;

$increment = \_\_::once(function() use (&$num) { return $num++; });

$increment();

$increment();

echo $num; // 1

**after** \_\_::after(count, function)

Creates a version of the function that will only run after being called **count** times.

$func = \_\_::after(3, function() { return 'x'; });

$func(); //

$func(); //

$func(); // 'x'

**wrap** \_\_::wrap(function, wrapper)

Wraps the **function** inside the **wrapper** function, passing it as the first argument. Lets the **wrapper** execute code before and/or after the **function** runs.

$hello = function($name) { return 'hello: ' . $name; };

$hi = \_\_::wrap($hello, function($func) {

return 'before, ' . $func('moe') . ', after';

});

$hi(); // 'before, hello: moe, after'

**compose** \_\_::compose(\*functions)

Returns the composition of the **functions**, where the return value is passed to the following function.

$greet = function($name) { return 'hi: ' . $name; };

$exclaim = function($statement) { return $statement . '!'; };

$welcome = \_\_::compose($exclaim, $greet);

$welcome('moe'); // 'hi: moe!'

**Objects**

**keys** \_\_::keys(object)

Get the keys

\_\_::keys((object) array('name'=>'moe', 'age'=>40)); // array('name', 'age')

**values** \_\_::values(object)

Get the values

\_\_::values((object) array('name'=>'moe', 'age'=>40)); // array('moe', 40)

**functions** \_\_::functions(object)*Alias:****methods***

Get the names of functions available to the object

class Stooge {

public function getName() { return 'moe'; }

public function getAge() { return 40; }

}

$stooge = new Stooge;

\_\_::functions($stooge); // array('getName', 'getAge')

**extend** \_\_::extend(destination, \*sources)

Copy all properties from the **source** objects into the **destination** object. Copying happens in order, so rightmost sources have override power.

\_\_::extend((object) array('name'=>'moe'), (object) array('age'=>50));

// (object) array('name'=>'moe', 'age'=>50)

**defaults** \_\_::defaults(object, \*defaults)

Returns the object with any missing values filled in using the defaults. Once a default is applied for a given property, it will not be overridden by following defaults.

$food = (object) array('dairy'=>'cheese');

$defaults = (object) array('meat'=>'bacon');

\_\_::defaults($food, $defaults); // (object) array('dairy'=>'cheese', 'meat'=>'bacon');

**clon** \_\_::clon(object)

Returns a shallow copy of the object. This function is called 'clone' in Underscore.js, but was renamed to 'clon' in Underscore.php because 'clone' is a reserved keyword in PHP.

$stooge = (object) array('name'=>'moe');

\_\_::clon($stooge); // (object) array('name'=>'moe');

**tap** \_\_::tap(object, interceptor)

Invokes the **interceptor** on the **object**, then returns the object. Useful for performing intermediary operations on the object.

$interceptor = function($obj) { return $obj \* 2; };

\_\_::chain(array(1, 2, 3))->max()

->tap($interceptor)

->value(); // 6

**has** \_\_::has(object, key)

Does the object have this key?

\_\_::has((object) array('a'=>1, 'b'=>2, 'c'=>3), 'b'); // true

**isEqual** \_\_::isEqual(object, other)

Are these items equal? Uses === equality testing. Objects tested using values.

$stooge = (object) array('name'=>'moe');

$clon = \_\_::clon($stooge);

$stooge === $clon; // false

\_\_::isEqual($stooge, $clon); // true

**isEmpty** \_\_::isEmpty(object)

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

$stooge = (object) array('name'=>'moe');

\_\_::isEmpty($stooge); // false

\_\_::isEmpty(new StdClass); // true

\_\_::isEmpty((object) array()); // true

**isObject** \_\_::isObject(object)

Returns true if passed an object.

\_\_::isObject((object) array(1, 2)); // true

\_\_::isObject(new StdClass); // true

**isArray** \_\_::isArray(object)

Returns true if passed an array.

\_\_::isArray(array(1, 2)); // true

\_\_::isArray((object) array(1, 2)); // false

**isFunction** \_\_::isFunction(object)

Returns true if passed a function.

\_\_::isFunction(function() {}); // true

\_\_::isFunction('trim'); // false

**isString** \_\_::isString(object)

Returns true if passed a string.

\_\_::isString('moe'); // true

\_\_::isString(''); // true

**isNumber** \_\_::isNumber(object)

Returns true if passed a number.

\_\_::isNumber(1); // true

\_\_::isNumber(2.5); // true

\_\_::isNumber('5'); // false

**isBoolean** \_\_::isBoolean(object)

Returns true if passed a boolean.

\_\_::isBoolean(null); // false

\_\_::isBoolean(true); // true

\_\_::isBoolean(0); // false

**isDate** \_\_::isDate(object)

Returns true if passed a DateTime object

\_\_::isDate(null); // false

\_\_::isDate('2011-06-09 01:02:03'); // false

\_\_::isDate(new DateTime); // true

**isNaN** \_\_::isNaN(object)

Returns true if value is NaN

\_\_::isNaN(null); // false

\_\_::isNaN(acos(8)); // true

**isNull** \_\_::isNull(object)

Returns true if value is null

\_\_::isNull(null); // true

\_\_::isNull(false); // false

**Utility**

**identity** \_\_::identity(value)

Returns the same value passed as the argument

$moe = array('name'=>'moe');

$moe === \_\_::identity($moe); // true

**times** \_\_::times(n, iterator)

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

\_\_::times(3, function() { echo 'a'; }); // 'aaa'

**mixin** \_\_::mixin(array)

Extend Underscore.php with your own functions.

\_\_::mixin(array(

'capitalize'=> function($string) { return ucwords($string); },

'yell' => function($string) { return strtoupper($string); }

));

\_\_::capitalize('moe'); // 'Moe'

\_\_::yell('moe'); // 'MOE'

**uniqueId** \_\_::uniqueId([prefix])

Generate a globally unique id.

\_\_::uniqueId(); // 0

\_\_::uniqueId('stooge\_'); // 'stooge\_1'

\_\_::uniqueId(); // 2

**escape** \_\_::escape(html)

Escapes the string.

\_\_::escape('Curly, Larry & Moe'); // 'Curly, Larry &amp; Moe'

**template** \_\_::template(templateString, [context])

Compile templates into functions that can be evaluated for rendering. Templates can interpolate variables and execute arbitrary PHP code.

$compiled = \_\_::template('hello: <%= $name %>');

$compiled(array('name'=>'moe')); // 'hello: moe'

$list = '<% \_\_::each($people, function($name) { %> <li><%= $name %></li> <% }); %>';

\_\_::template($list, array('people'=>array('moe', 'curly', 'larry')));

// '<li>moe</li><li>curly</li><li>larry</li>'

**Single vs. double quotes**

Note: if your template strings include variables, wrap your template strings in single quotes, not double quotes. Wrapping in double quotes will cause your variables to be interpolated prior to entering the template function.

// Correct

$compiled = \_\_::template('hello: <%= $name %>');

// Incorrect

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

**Custom delimiters**

You can set custom delimiters (for instance, Mustache style) by calling \_\_::templateSettings() and passing interpolate and/or evaluate values:

// Mustache style

\_\_::templateSettings(array(

'interpolate' => '/\{\{(.+?)\}\}/'

));

$mustache = \_\_::template('Hello {{$planet}}!');

$mustache(array('planet'=>'World')); // "Hello World!"

**Chaining**

**chain** \_\_::chain(item)

Returns a wrapped object. Methods will return the object until you call value()

// filter and reverse the numbers

$numbers = array(1, 2, 3, 4, 5, 6, 7, 8, 9, 10);

$result = \_\_::chain($numbers)->select(function($n) { return $n < 5; })

->reject(function($n) { return $n === 3; })

->sortBy(function($n) { return -$n; })

->value(); // array(4, 2, 1)

**value** \_\_(obj)->value()

Extracts the value of a wrapped object.

\_\_(array(1, 2, 3))->value(); // array(1, 2, 3)