UNDERSCORE.JS

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], alert);


=> alerts each number in turn...


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


=> alerts each number value in turn...
```

map_.map(list, iterator, [context]) *Alias:* ***collect***
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)`.

```
_.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]
```

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**. The iterator is passed four arguments: the `memo`, then the `value`and `index` (or key) of the iteration, and finally a reference to the entire `list`.

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


=> 6
```

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.

```
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, 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.

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


=> 2
```

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.

```
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**.

```
_.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, iterator, [context])`
Returns the values in **list** without the elements that the truth test (**iterator**) 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, iterator, [context])` *Alias:* ***all***
Returns *true* if all of the values in the **list** pass the **iterator** truth test. Delegates to the native method **every**, if present.

```
_.every([true, 1, null, 'yes'], _.identity);

=> false
```

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

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

```
=> true
```

contains_.contains(list, value) *Alias:* ***include***
Returns *true* if the **value** is present in the **list**. Uses **indexOf** internally, if **list** is an Array.

```
_.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, [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.

```
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, [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.

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

_.min(numbers);

=> 2
```

sortBy `_.sortBy(list, iterator, [context])`
Returns a sorted copy of **list**, ranked in ascending order by the results of running each value through **iterator**. Iterator 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]
```

groupBy `_.groupBy(list, iterator, [context])`
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.

```
_.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"]}
```

countBy `_.countBy(list, iterator, [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]
```

toArray_`.toArray(list)`
Converts the **list** (anything that can be iterated over), into a real Array. 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
```

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], [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.

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


=> [1, 2, 3, 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.

```
_.zip(['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.

```
_.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**. 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 ... 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. Uses the native **lastIndexOf** function if possible. Pass **fromIndex** to start your search at a given index.

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


=> 4
```

sortedIndex `_.sortedIndex(list, value, [iterator], [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 **iterator** is passed, it will be used to compute the sort ranking of each value, including the **value** you pass.

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


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

```
_.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**.

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

```
var buttonView = {


  label   : 'underscore',


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

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

};

_.bindAll(buttonView);

jQuery('#underscore_button').bind('click', buttonView.onClick);

=> When the button is clicked, this.label will have the correct value...
```

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.

```
var add = function(a, b) { return a + b; };

add5 = _.partial(add, 5);

add5(10);

=> 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.

```
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)`
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.

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


$(window).scroll(throttled);
```

debounce `_.debounce(function, wait, [immediate])`
Creates and returns a new debounced version of the passed function that 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** parameter 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.
```

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'
```

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 + "!"; };

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

welcome('moe');

=> 'hi: moe!'
```

Object Functions

keys_.keys(object)
Retrieve all the names of the **object**'s properties.

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

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

values_.values(object)
Return all of the values of the **object**'s properties.

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

```
=> [1, 2, 3]
```

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"};
```

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" ...
```

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}
```

pick_.pick(object, *keys)
Return a copy of the **object**, filtered to only have values for the whitelisted **keys** (or array of valid keys).

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


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

omit `_.omit(object, *keys)`
Return a copy of the **object**, filtered to omit the blacklisted **keys** (or array of keys).

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


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

defaults `_.defaults(object, *defaults)`
Fill in null and undefined properties in **object** with values from the **defaults** objects, and return the **object**. As soon as the property is filled, further defaults will have no effect.

```
var iceCream = {flavor : "chocolate"};


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


=> {flavor : "chocolate", 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.

```
_.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
```

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

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

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

moe == clone;

=> false

_.isEqual(moe, clone);

=> true
```

isEmpty `_.isEmpty(object)`
Returns *true* if **object** contains no values.

```
_.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
```

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*.

```
_.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 iterator.

```
var moe = {name : 'moe'};

moe === _.identity(moe);

=> true
```

times `_.times(n, iterator, [context])`
Invokes the given iterator function **n** times. Each invocation of **iterator** is called with an `index` argument.
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"
```

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;`, `&#x27;`, and `&#x2F;` with their unescaped counterparts.

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


=> "Curly, Larry & Moe"
```

result _.result(object, property)
If the value of the named property is a function then invoke it; otherwise, return it.

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


_.result(object, 'cheese');


=> "crumpets"


_.result(object, 'stuff');


=> "nonsense"
```

template_.template(templateString, [data], [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 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 **data** object that has properties corresponding to the template's free variables. If you're writing a one-off, you can pass the **data** object as the second parameter to **template** in order to render immediately instead of returning a template function.
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 list = "<% _.each(people, function(name) { %> <li><%= name %></li> <% }); %>";

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

=> "<li>moe</li><li>curly</li><li>larry</li>"



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 %>", {answer: 'no'}, {variable: 'data'});


=> "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, 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.

```
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 used.

```
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]
```