

---

## Lume

A collection of functions for Lua, geared towards game development.

### Installation

The lume.lua file should be dropped into an existing project and required by it:

```
1 lume = require "lume"
```

### Function Reference

**lume.clamp(x, min, max)** Returns the number `x` clamped between the numbers `min` and `max`

**lume.round(x [, increment])** Rounds `x` to the nearest integer; rounds away from zero if we're mid-way between two integers. If `increment` is set then the number is rounded to the nearest increment.

```
1 lume.round(2.3) -- Returns 2
2 lume.round(123.4567, .1) -- Returns 123.5
```

**lume.sign(x)** Returns 1 if `x` is 0 or above, returns -1 when `x` is negative.

**lume.lerp(a, b, amount)** Returns the linearly interpolated number between `a` and `b`, `amount` should be in the range of 0 - 1; if `amount` is outside of this range it is clamped.

```
1 lume.lerp(100, 200, .5) -- Returns 150
```

**lume.smooth(a, b, amount)** Similar to `lume.lerp()` but uses cubic interpolation instead of linear interpolation.

**lume.pingpong(x)** Ping-pongs the number `x` between 0 and 1.

**lume.distance(x1, y1, x2, y2 [, squared])** Returns the distance between the two points. If `squared` is true then the squared distance is returned – this is faster to calculate and can still be used when comparing distances.

---

**lume.angle(x1, y1, x2, y2)** Returns the angle between the two points.

**lume.vector(angle, magnitude)** Given an **angle** and **magnitude**, returns a vector.

```
1 local x, y = lume.vector(0, 10) -- Returns 10, 0
```

**lume.random([a [, b]])** Returns a random number between **a** and **b**. If only **a** is supplied a number between 0 and **a** is returned. If no arguments are supplied a random number between 0 and 1 is returned.

**lume.randomchoice(t)** Returns a random value from array **t**. If the array is empty an error is raised.

```
1 lume.randomchoice({true, false}) -- Returns either true or false
```

**lume.weightedchoice(t)** Takes the argument table **t** where the keys are the possible choices and the value is the choice's weight. A weight should be 0 or above, the larger the number the higher the probability of that choice being picked. If the table is empty, a weight is below zero or all the weights are 0 then an error is raised.

```
1 lume.weightedchoice({ ["cat"] = 10, ["dog"] = 5, ["frog"] = 0 })
2 -- Returns either "cat" or "dog" with "cat" being twice as likely to be
   chosen.
```

**lume.isarray(x)** Returns **true** if **x** is an array – the value is assumed to be an array if it is a table which contains a value at the index 1. This function is used internally and can be overridden if you wish to use a different method to detect arrays.

**lume.push(t, ...)** Pushes all the given values to the end of the table **t** and returns the pushed values. Nil values are ignored.

```
1 local t = { 1, 2, 3 }
2 lume.push(t, 4, 5) -- `t` becomes { 1, 2, 3, 4, 5 }
```

**lume.remove(t, x)** Removes the first instance of the value **x** if it exists in the table **t**. Returns **x**.

```
1 local t = { 1, 2, 3 }
2 lume.remove(t, 2) -- `t` becomes { 1, 3 }
```

---

**lume.clear(t)** Nils all the values in the table `t`, this renders the table empty. Returns `t`.

```
1 local t = { 1, 2, 3 }
2 lume.clear(t) -- `t` becomes {}
```

**lume.extend(t, ...)** Copies all the fields from the source tables to the table `t` and returns `t`. If a key exists in multiple tables the right-most table's value is used.

```
1 local t = { a = 1, b = 2 }
2 lume.extend(t, { b = 4, c = 6 }) -- `t` becomes { a = 1, b = 4, c = 6 }
```

**lume.shuffle(t)** Returns a shuffled copy of the array `t`.

**lume.sort(t [, comp])** Returns a copy of the array `t` with all its items sorted. If `comp` is a function it will be used to compare the items when sorting. If `comp` is a string it will be used as the key to sort the items by.

```
1 lume.sort({ 1, 4, 3, 2, 5 }) -- Returns { 1, 2, 3, 4, 5 }
2 lume.sort({ {z=2}, {z=3}, {z=1} }, "z") -- Returns { {z=1}, {z=2}, {z=3} }
3 lume.sort({ 1, 3, 2 }, function(a, b) return a > b end) -- Returns { 3, 2, 1 }
```

**lume.array(...)** Iterates the supplied iterator and returns an array filled with the values.

```
1 lume.array(string.gmatch("Hello world", "%a+")) -- Returns {"Hello", "world"}
```

**lume.each(t, fn, ...)** Iterates the table `t` and calls the function `fn` on each value followed by the supplied additional arguments; if `fn` is a string the method of that name is called for each value. The function returns `t` unmodified.

```
1 lume.each({1, 2, 3}, print) -- Prints "1", "2", "3" on separate lines
2 lume.each({a, b, c}, "move", 10, 20) -- Does x:move(10, 20) on each value
```

**lume.map(t, fn)** Applies the function `fn` to each value in table `t` and returns a new table with the resulting values.

```
1 lume.map({1, 2, 3}, function(x) return x * 2 end) -- Returns {2, 4, 6}
```

---

**lume.all(t [, fn])** Returns true if all the values in **t** table are true. If a **fn** function is supplied it is called on each value, true is returned if all of the calls to **fn** return true.

```
1 lume.all({1, 2, 1}, function(x) return x == 1 end) -- Returns false
```

**lume.any(t [, fn])** Returns true if any of the values in **t** table are true. If a **fn** function is supplied it is called on each value, true is returned if any of the calls to **fn** return true.

```
1 lume.any({1, 2, 1}, function(x) return x == 1 end) -- Returns true
```

**lume.reduce(t, fn [, first])** Applies **fn** on two arguments cumulative to the items of the array **t**, from left to right, so as to reduce the array to a single value. If a **first** value is specified the accumulator is initialised to this, otherwise the first value in the array is used. If the array is empty and no **first** value is specified an error is raised.

```
1 lume.reduce({1, 2, 3}, function(a, b) return a + b end) -- Returns 6
```

**lume.unique(t)** Returns a copy of the **t** array with all the duplicate values removed.

```
1 lume.unique({2, 1, 2, "cat", "cat"}) -- Returns {1, 2, "cat"}
```

**lume.filter(t, fn [, retainkeys])** Calls **fn** on each value of **t** table. Returns a new table with only the values where **fn** returned true. If **retainkeys** is true the table is not treated as an array and retains its original keys.

```
1 lume.filter({1, 2, 3, 4}, function(x) return x % 2 == 0 end) -- Returns
  {2, 4}
```

**lume.reject(t, fn [, retainkeys])** The opposite of **lume.filter()**: Calls **fn** on each value of **t** table; returns a new table with only the values where **fn** returned false. If **retainkeys** is true the table is not treated as an array and retains its original keys.

```
1 lume.reject({1, 2, 3, 4}, function(x) return x % 2 == 0 end) -- Returns
  {1, 3}
```

**lume.merge(...)** Returns a new table with all the given tables merged together. If a key exists in multiple tables the right-most table's value is used.

```
1 lume.merge({a=1, b=2, c=3}, {c=8, d=9}) -- Returns {a=1, b=2, c=8, d=9}
```

---

**lume.concat(...)** Returns a new array consisting of all the given arrays concatenated into one.

```
1 lume.concat({1, 2}, {3, 4}, {5, 6}) -- Returns {1, 2, 3, 4, 5, 6}
```

**lume.find(t, value)** Returns the index/key of `value` in `t`. Returns `nil` if that value does not exist in the table.

```
1 lume.find({"a", "b", "c"}, "b") -- Returns 2
```

**lume.match(t, fn)** Returns the value and key of the value in table `t` which returns true when `fn` is called on it. Returns `nil` if no such value exists.

```
1 lume.match({1, 5, 8, 7}, function(x) return x % 2 == 0 end) -- Returns
  8, 3
```

**lume.count(t [, fn])** Counts the number of values in the table `t`. If a `fn` function is supplied it is called on each value, the number of times it returns true is counted.

```
1 lume.count({a = 2, b = 3, c = 4, d = 5}) -- Returns 4
2 lume.count({1, 2, 4, 6}, function(x) return x % 2 == 0 end) -- Returns
  3
```

**lume.slice(t [, i [, j]])** Mimics the behaviour of Lua's `string.sub`, but operates on an array rather than a string. Creates and returns a new array of the given slice.

```
1 lume.slice({"a", "b", "c", "d", "e"}, 2, 4) -- Returns {"b", "c", "d"}
```

**lume.first(t [, n])** Returns the first element of an array or nil if the array is empty. If `n` is specified an array of the first `n` elements is returned.

```
1 lume.first({"a", "b", "c"}) -- Returns "a"
```

**lume.last(t [, n])** Returns the last element of an array or nil if the array is empty. If `n` is specified an array of the last `n` elements is returned.

```
1 lume.last({"a", "b", "c"}) -- Returns "c"
```

---

**lume.invert(t)** Returns a copy of the table where the keys have become the values and the values the keys.

```
1 lume.invert({a = "x", b = "y"}) -- returns {x = "a", y = "b"}
```

**lume.pick(t, ...)** Returns a copy of the table filtered to only contain values for the given keys.

```
1 lume.pick({ a = 1, b = 2, c = 3 }, "a", "c") -- Returns { a = 1, c = 3 }
}
```

**lume.keys(t)** Returns an array containing each key of the table.

**lume.clone(t)** Returns a shallow copy of the table `t`.

**lume.fn(fn, ...)** Creates a wrapper function around function `fn`, automatically inserting the arguments into `fn` which will persist every time the wrapper is called. Any arguments which are passed to the returned function will be inserted after the already existing arguments passed to `fn`.

```
1 local f = lume.fn(print, "Hello")
2 f("world") -- Prints "Hello world"
```

**lume.once(fn, ...)** Returns a wrapper function to `fn` which takes the supplied arguments. The wrapper function will call `fn` on the first call and do nothing on any subsequent calls.

```
1 local f = lume.once(print, "Hello")
2 f() -- Prints "Hello"
3 f() -- Does nothing
```

**lume.memoize(fn)** Returns a wrapper function to `fn` where the results for any given set of arguments are cached. `lume.memoize()` is useful when used on functions with slow-running computations.

```
1 fib = lume.memoize(function(n) return n < 2 and n or fib(n-1) + fib(n-2) end)
```

**lume.combine(...)** Creates a wrapper function which calls each supplied argument in the order they were passed to `lume.combine()`; nil arguments are ignored. The wrapper function passes its own arguments to each of its wrapped functions when it is called.

---

```
1 local f = lume.combine(function(a, b) print(a + b) end,
2                       function(a, b) print(a * b) end)
3 f(3, 4) -- Prints "7" then "12" on a new line
```

**lume.call(fn, ...)** Calls the given function with the provided arguments and returns its values. If **fn** is **nil** then no action is performed and the function returns **nil**.

```
1 lume.call(print, "Hello world") -- Prints "Hello world"
```

**lume.time(fn, ...)** Inserts the arguments into function **fn** and calls it. Returns the time in seconds the function **fn** took to execute followed by **fn**'s returned values.

```
1 lume.time(function(x) return x end, "hello") -- Returns 0, "hello"
```

**lume.lambda(str)** Takes a string lambda and returns a function. **str** should be a list of comma-separated parameters, followed by **->**, followed by the expression which will be evaluated and returned.

```
1 local f = lume.lambda "x,y -> 2*x+y"
2 f(10, 5) -- Returns 25
```

**lume.serialize(x)** Serializes the argument **x** into a string which can be loaded again using **lume.deserialize()**. Only booleans, numbers, tables and strings can be serialized. Circular references will result in an error; all nested tables are serialized as unique tables.

```
1 lume.serialize({a = "test", b = {1, 2, 3}, false})
2 -- Returns "{[1]=false,[\"a\"]=\"test\",[\"b\"]={ [1]=1,[2]=2,[3]=3,},}"
```

**lume.deserialize(str)** Deserializes a string created by **lume.serialize()** and returns the resulting value. This function should not be run on an untrusted string.

```
1 lume.deserialize("{1, 2, 3}") -- Returns {1, 2, 3}
```

**lume.split(str [, sep])** Returns an array of the words in the string **str**. If **sep** is provided it is used as the delimiter, consecutive delimiters are not grouped together and will delimit empty strings.

```
1 lume.split("One two three") -- Returns {"One", "two", "three"}
2 lume.split("a,b,,c", ",") -- Returns {"a", "b", "", "c"}
```

---

**lume.trim(str [, chars])** Trims the whitespace from the start and end of the string `str` and returns the new string. If a `chars` value is set the characters in `chars` are trimmed instead of whitespace.

```
1 lume.trim(" Hello ") -- Returns "Hello"
```

**lume.wordwrap(str [, limit])** Returns `str` wrapped to `limit` number of characters per line, by default `limit` is 72. `limit` can also be a function which when passed a string, returns `true` if it is too long for a single line.

```
1 -- Returns "Hello world\nThis is a\nshort string"
2 lume.wordwrap("Hello world. This is a short string", 14)
```

**lume.format(str [, vars])** Returns a formatted string. The values of keys in the table `vars` can be inserted into the string by using the form `"{key}"` in `str`; numerical keys can also be used.

```
1 lume.format("{b} hi {a}", {a = "mark", b = "Oh"}) -- Returns "Oh hi
mark"
2 lume.format("Hello {1}!", {"world"}) -- Returns "Hello world!"
```

**lume.trace(...)** Prints the current filename and line number followed by each argument separated by a space.

```
1 -- Assuming the file is called "example.lua" and the next line is 12:
2 lume.trace("hello", 1234) -- Prints "example.lua:12: hello 1234"
```

**lume.dostring(str)** Executes the lua code inside `str`.

```
1 lume.dostring("print('Hello!')") -- Prints "Hello!"
```

**lume.uuid()** Generates a random UUID string; version 4 as specified in RFC 4122.

**lume.hotswap(modname)** Reloads an already loaded module in place, allowing you to immediately see the effects of code changes without having to restart the program. `modname` should be the same string used when loading the module with `require()`. In the case of an error the global environment is restored and `nil` plus an error message is returned.

```
1 lume.hotswap("lume") -- Reloads the lume module
2 assert(lume.hotswap("inexistent_module")) -- Raises an error
```



---

**lume.ripairs(t)** Performs the same function as `ipairs()` but iterates in reverse; this allows the removal of items from the table during iteration without any items being skipped.

```
1 -- Prints "3->c", "2->b" and "1->a" on separate lines
2 for i, v in lume.ripairs({ "a", "b", "c" }) do
3   print(i .. "->" .. v)
4 end
```

**lume.color(str [, mul])** Takes color string `str` and returns 4 values, one for each color channel (`r`, `g`, `b` and `a`). By default the returned values are between 0 and 1; the values are multiplied by the number `mul` if it is provided.

```
1 lume.color("#ff0000") -- Returns 1, 0, 0, 1
2 lume.color("rgba(255, 0, 255, .5)") -- Returns 1, 0, 1, .5
3 lume.color("#00ffff", 256) -- Returns 0, 256, 256, 256
4 lume.color("rgb(255, 0, 0)", 256) -- Returns 256, 0, 0, 256
```

**lume.chain(value)** Returns a wrapped object which allows chaining of lume functions. The function `result()` should be called at the end of the chain to return the resulting value.

```
1 lume.chain({1, 2, 3, 4})
2   :filter(function(x) return x % 2 == 0 end)
3   :map(function(x) return -x end)
4   :result() -- Returns { -2, -4 }
```

The table returned by the `lume` module, when called, acts in the same manner as calling `lume.chain()`.

```
1 lume({1, 2, 3}):each(print) -- Prints 1, 2 then 3 on separate lines
```

## Iteratee functions

Several lume functions allow a `table`, `string` or `nil` to be used in place of their iteratee function argument. The functions that provide this behaviour are: `map()`, `all()`, `any()`, `filter()`, `reject()`, `match()` and `count()`.

If the argument is `nil` then each value will return itself.

```
1 lume.filter({ true, true, false, true }, nil) -- { true, true, true }
```

If the argument is a `string` then each value will be assumed to be a table, and will return the value of the key which matches the string.

---

```
1 local t = {{ z = "cat" }, { z = "dog" }, { z = "owl" }}
2 lume.map(t, "z") -- Returns { "cat", "dog", "owl" }
```

If the argument is a **table** then each value will return **true** or **false**, depending on whether the values at each of the table's keys match the collection's value's values.

```
1 local t = {
2   { age = 10, type = "cat" },
3   { age = 8,  type = "dog" },
4   { age = 10, type = "owl" },
5 }
6 lume.count(t, { age = 10 }) -- returns 2
```

## License

This library is free software; you can redistribute it and/or modify it under the terms of the MIT license. See LICENSE for details.