On This Page

- **MoonScript Standard Library**
  - **Printing Functions**
    - `p(arg)`
  - **Table Functions**
    - `run_with_scope(fn, scope, [args...])`
    - `defaulttbl([tbl,] fn)`
    - `extend(arg1, arg2, [rest...])`
    - `copy(tbl)`
  - **Class/Object Functions**
    - `is_object(value)`
    - `type(value)`
    - `bind_methods(obj)`
    - `mixin(obj, class, [args...])`
    - `mixin_object(obj, other_obj, method_names)`
    - `mixin_table(a, b, [names])`
  - **Misc Functions**
    - `fold(items, fn)`
  - **Debug Functions**
    - `debug.upvalue(fn, key[, value])`

All Pages

- **Language Guide**
- **Standard Library**
- **Command Line Tools**
- **Compiler API**
The MoonScript installation comes with a small kernel of functions that can do various common things.

The entire library is currently contained in a single object. We can bring this library requiring "moon".

```
require "moon"
-- `moon.p` is the debug printer
moon.p { hello: "world" }
```

If you prefer to just inject all of the functions into the current scope, you can require "moon.all"

```
require "moon.all"
p { hello: "world" }
```

All of the functions are compatible with Lua in addition to MoonScript, but more sense in the context of MoonScript.
This is an overview of all the included functions. All of the examples assume has been included with `require "moon.all"`. 
Printing Functions

\[ p(\text{arg}) \]

Prints a formatted version of an object. Excellent for inspecting the conten
Table Functions

run_with_scope(fn, scope, [args...])

Mutates the environment of function fn and runs the function with any extra arguments provided.

Returns the result of the function.

The environment of the function is set to a new table whose metatable will use values. scope must be a table. If scope does not have an entry for a value, it falls back on the original environment.

```lua
my_env = {
  secret_function: -> print "shhh this is secret"
  say_hi: -> print "hi there!"
}

say_hi = -> print "I am a closure"

fn = ->
  secret_function!
  say_hi!

run_with_scope fn, my_env
```

Note that any closure values will always take precedence against global variables. In the example above, the say_hi in the environment has been shadowed by the local variable say_hi.

defaulttbl([tbl,] fn)

Sets the __index of table tbl to use the function fn to generate table values when a missing key is looked up.

```lua
extend(arg1, arg2, [rest...])
```
Chains together a series of tables by their metatable’s `__index` property. Overwrites the metatable of all objects except for the last with a new table whose `__index` is set to the next table.

Returns the first argument.

```lua
a = { hello: "world" }
b = { okay: "sure" }

extend a, b

print a.okay
```

`copy(tbl)`

Creates a shallow copy of a table, equivalent to:

```lua
copy = (arg) -> {k,v for k,v in pairs self}
```
Class/Object Functions

**is_object(value)**

Returns true if `value` is an instance of a MoonScript class, false otherwise.

**type(value)**

If `value` is an instance of a MoonScript class, then return it’s class object. Otherwise, return the result of calling Lua’s type method.

```moonscript
class MyClass
    nil

x = MyClass!
assert type(x) == MyClass
```

**bind_methods(obj)**

Takes an instance of an object, returns a proxy to the object whose method providing self as the first argument.

```moonscript
obj = SomeClass!
bound_obj = bind_methods obj

-- following have the same effect
obj\hello!
bound_obj.hello!
```

It lazily creates and stores in the proxy table the bound methods when the

**mixin(obj, class, [args...])**
Copies the methods of a class \texttt{cls} into the table \texttt{obj}, then calls the constructor of the class with \texttt{obj} as the receiver.

In this example we add the functionality of \texttt{First} to an instance of \texttt{Second} without ever instancing \texttt{First}.

```python
class First
    new: (@var) =>
        show_var: => print "var is:", @var

class Second
    new: =>
        mixin self, First, "hi"

a = Second!
a\show_var!
```

Be wary of name collisions when mixing in other classes, names will be overwritten.

```python
mixin_object(obj, other_obj, method_names)
```

Inserts into \texttt{obj} methods from \texttt{other_obj} whose names are listed in \texttt{method_names} are bound methods that will run with \texttt{other_obj} as the receiver.

```python
class List
    add: (item) => print "adding to", self
    remove: (item) => print "removing from", self

class Encapsulation
    new: =>
        @list = List!
        mixin_object self, @list, {"add", "remove"}

e = Encapsulation!
e.add "something"
```

```python
mixin_table(a, b, [names])
```
Copies the elements of table b into table a. If names is provided, then o
Calls function `fn` repeatedly with the accumulated value and the current `items`. The accumulated value is the result of the last call to `fn`, or, in the base case, the first value. The current value is the value being iterated over starting with the second `items` is a normal array table.

For example, to sum all numbers in a list:

```
numbers = {4, 3, 5, 6, 7, 2, 3}
sum = fold numbers, (a, b) -> a + b
```
Debug Functions

d debug.upvalue(fn, key[, value])

Gets or sets the value of an upvalue for a function by name.

Generated on Thu Jun 19 00:40:22 2014; MoonScript v0.2.6