MoonScript 0.5.0 - Language Guide

On This Page

- The Language
 - Whitespace
 - Assignment
 - Update Assignment
 - Comments
 - Literals & Operators
 - Function Literals
 - Fat Arrows
 - Argument Defaults
 - Considerations
 - Multi-line arguments
 - Table Literals
 - Comprehensions
 - List Comprehensions
 - Table Comprehensions
 - Slicing
 - String Interpolation
 - For Loop
 - While Loop
 - Continue
 - Conditionals
 - With Assignment
 - Line Decorators
 - Switch
 - Object Oriented Programming
 - Inheritance
 - Super
 - Types
 - Class Objects
 - Class Variables
 - Class Declaration Statements

- and @ Values
- Class Expressions
- Anonymous classes
- Export Statement
 - Export All & Export Proper
- Local Statement
- Import Statement
- With Statement
- **Do**
- Destructuring Assignment
 - Destructuring In Other Places
- Function Stubs
- The Using Clause; Controlling Destructive Assignment
- Misc.
 - Implicit Returns on Files
 - Writing Modules
- License (MIT)

All Pages

- Language Guide
- Standard Library
- Command Line Tools
- Compiler API

MoonScript is a programming language that compiles to Lua. This guide e basic familiarity with Lua. For each code snippet below, the MoonScript is Lua is on right right.

This is the official language reference manual, installation directions and the http://moonscript.org.

The Language

Whitespace

MoonScript is a whitespace sensitive language. This means that instead c and r) to delimit sections of code we use line-breaks and indentation.

This means that how you indent your code is important. Luckily MoonScrij it but only requires that you be consistent.

An indent must be at least 1 space or 1 tab, but you can use as many as y snippets on this page will use two spaces.

Note: Should you happen to mix tabs and spaces, a tab is equivalent to telling you this though because you should never do it.

Assignment

Assigning to an undeclared name will cause it to be declared as a new loc dynamically typed so you can assign any value to any variable. You can a values at once just like Lua:

MoonScript		Lua
	hello	
	world	
	a	
	, b	
	′ C	
	= 1	
	2	
	, 3 hello	
	=	
	123 uses the existing varia	able

If you wish to create a global variable it must be done using the export ke

The local keyword can be used to forward declare a variable, or shadow

+=, -=, /=, *=, %=, ..=, or=, and= operators have been added for upd same time. They are aliases for their expanded equivalents.

MoonScript		Lua
	<pre>x = 0 x += 10 s = " hello " s= " world " b = false b and = true or false</pre>	

Comments

Like Lua, comments start with _-_ and continue to the end of the line. Con the output.

MoonScript		Lua
	I am a comment	

Literals & Operators

All of the primitive literals in Lua can be used. This applies to numbers, str

All of Lua's binary and unary operators are available. Additionally **[15]** is as

Unlike Lua, Line breaks are allowed inside of single and double quote strin sequence:

MoonScript	Lua
	some_string = " Here is a string that has a line break in it. "

Function Literals

All functions are created using a function expression. A simple function is

MoonScript	Lua
	<pre>my_function = -> my_function () call the empty function</pre>

The body of the function can either be one statement placed directly after series of statements indented on the following lines:

MoonScript		Lua
	<pre>func_a = -> print " hello world " func_b = -> value = 100 print " The value: "</pre>	
	value	

If a function has no arguments, it can be called using the <u>__</u> operator, inst The <u>__</u> invocation is the preferred way to call functions with no arguments

MoonScript		Lua
	func_a ! func_b ()	

Functions with arguments can be created by preceding the arrow with a lis parentheses:

MoonScript		Lua
	sum	
	=	
	(
	Х	
	1	
	y)	
) ->	
	print	
	sum	
	п	
	1	



Functions can be called by listing the arguments after the name of an expl function. When chaining together function calls, the arguments are applied the left.

MoonScript		Lua
	<pre>Sum 10 ' 20 print sum 10 ' 20 a b C " a " ' ' b " ' ' C " "</pre>	

In order to avoid ambiguity in when calling functions, parentheses can also arguments. This is required here in order to make sure the right argument functions.

MoonScript		Lua
	print "	
	X: "	
	, Sum	
	(10	
	20	
) /	
	y: "	
	, sum	
	(30	
	40	
)	

There must not be any space between the opening parenthesis and the fu

Functions will coerce the last statement in their body into a return stateme return:

MoonScript		Lua
	C II M	
	(
	X	
	ý	
	x , y	

) -> x + y print " The sum is " , sum 10 20

And if you need to explicitly return, you can use the return keyword:

MoonScript		Lua
	SIIM	
	=	
	(
	X	
	ý	
)	
	->	
	return	
	Х	
	+	
	У	

Just like in Lua, functions can return multiple values. The last statement m separated by commas:

MoonScript		Lua
	mystery	
	=	
	(
	× ,	
	ý	
)	
	x	
	+	
	ý	
	×	
	-	
	y a	
	!	
	b =	
	_ mystery	
	10	
	, 20	
	20	

Fat Arrows

Because it is an idiom in Lua to send an object as the first argument when syntax is provided for creating functions which automatically includes a section

MoonScript		Lua
	6	
	TUNC =	
	(

```
num
)
=>
@value
+
num
```

Argument Defaults

It is possible to provide default values for the arguments of a function. An be empty if its value is <u>nil</u>. Any <u>nil</u> arguments that have a default value body of the function is run.

```
MoonScript
                                                      Lua
                       my_function
                       =
                       (
                       name
                       =
                       н
                       something
                       н
                       height
                       =
                       100
                       )
                       ->
                       print
                       Hello I am
                       н
                       1
                       name
```

print " My height is , height

An argument default value expression is evaluated in the body of the func argument declarations. For this reason default values have access to prev

MoonScript		Lua
	<pre>some_args = (x = 100 ' y = x + 1000) -> print</pre>	

х + У

Considerations

Because of the expressive parentheses-less way of calling functions, som place to avoid parsing ambiguity involving whitespace.

The minus sign plays two roles, a unary negation operator and a binary su Consider how the following examples compile:

MoonScript		Lua
	a	
	x -	
	10 b	
	= x	
	-	

10
10
C
=
X
y .
d
=
X
Z

The precedence of the first argument of a function call can be controlled u argument is a literal string. In Lua, it is common to leave off parentheses v a single string or table literal.

When there is no space between a variable and a string literal, the functio over any following expressions. No other arguments can be passed to the this way.

Where there is a space following a variable and a string literal, the function The string literal belongs to any following expressions (if they exist), which list.



```
func
"
hello
"
+
100
```

Multi-line arguments

When calling functions that take a large number of arguments, it is conver list over multiple lines. Because of the white-space sensitive nature of the taken when splitting up the argument list.

If an argument list is to be continued onto the next line, the current line mu following line must be indented more than the current indentation. Once in lines must be at the same level of indentation to be part of the argument li

MoonScript		Lua
	my_tunc 5	
	′ 4	
	3	
	' 8	
	' 9	
	/ 10 cool_func 1	
	2	
	, 3	
	/	

4
, 5
, 6
′ 7
' 8

This type of invocation can be nested. The level of indentation is used to c the arguments belong to.

MoonScript		Lua
MoonScript	<pre>my_func 5 '6 '7 '6 'another_func 6 '7 '8 '9 '1 '2 '5 '4</pre>	Lua

Because tables also use the comma as a delimiter, this indentation syntax be part of the argument list instead of being part of the table.



Although uncommon, notice how we can give a deeper indentation for fun we will be using a lower indentation further on.

MoonScript		Lua
	V	

= { my_func 1 , 2 , 3
4
, 5
, 5
, 6
7
}

The same thing can be done with other block level statements like condition indentation level to determine what statement a value belongs to:

MoonScript		Lua
	if func 1	
	· 2	
	, 3	
	/ 11	
	hello "	
	/ 11	
	world	
	print "	
	hello "	

print " I am inside if 11 if func 1 **′** 2 ′ 3 / 11 hello н. / 11 world н print " hello н. print " I am inside if

Table Literals

Like in Lua, tables are delimited in curly braces.

MoonScript		Lua
	<pre>some_values = { 1 '2 '3 '4 }</pre>	

Unlike Lua, assigning a value to a key in a table is done with : (instead c

MoonScript		Lua
	some_values = { name: " Bill , age: 200 ,	
	" favorite food "] :	

```
"
rice
"
}
```

The curly braces can be left off if a single table of key value pairs is being

MoonScript		Lua
	<pre>profile = height: " 4 feet " 'shoe_size: 13 'favorite_foods: { " ice cream " ' donuts " }</pre>	

Newlines can be used to delimit values instead of a comma (or both):

MoonScript		Lua
	values = { 1 2	



When creating a single line table literal, the curly braces can also be left o



legs: 4 ′tails: 1

The keys of a table literal can be language keywords without being escape

MoonScript		Lua
	<pre>tbl = { do: " something " end: " hunger " }</pre>	

If you are constructing a table out of variables and wish the keys to be the names, then the : prefix operator can be used:

MoonScript	Lua

hair = н golden " height = 200 person = { :hair , :height shoe_size: 40 } print_table :hair ; height

If you want the key of a field in the table to to be result of an expression, the just like in Lua. You can also use a string literal directly as a key, leaving o is useful if your key has any special characters.

MoonScript		Lua
	t	
	=	

{
[
1
+
2
]
...
hello
"
hello world
"
true
}

Comprehensions

Comprehensions provide a convenient syntax for constructing a new table existing object and applying an expression to its values. There are two kin comprehensions and table comprehensions. They both produce Lua table accumulate values into an array-like table, and *table comprehensions* let y value on each iteration.

List Comprehensions

The following creates a copy of the *items* table but with all the values doul

MoonScript		Lua
	<pre>items = { 1 ' 2 ' 3 ' 4 } doubled = [item * 2 for i ' item in ipairs items]</pre>	Lua

The items included in the new table can be restricted with a when clause:

MoonScript		Lua
	<pre>iter = ipairs items slice = [item for i</pre>	
	, item in iter	

when
i
>
1
and
i
<
3
]</pre>

Because it is common to iterate over the values of a numerically indexed t introduced. The doubled example can be rewritten as:

MoonScript		Lua
	doubled	
	=	

[item * for item in * items] The for and when clauses can be chained as much as desired. The only comprehension has at least one for clause.

Using multiple for clauses is the same as using nested loops:

MoonScript		Lua
MoonScript	<pre>x_coords = { 4 '5 '6 '7 } y_coords = { 9 '2 '3 } points = [{ x ' </pre>	Lua
	y } for x in	
	x_coords for y in *	
]	

MoonScript Lua evens = [i for i = 1 , 100 when i % 2 == 0]

Numeric for loops can also be used in comprehensions:

Table Comprehensions

The syntax for table comprehensions is very similar, only differing by using values from each iteration.

This example makes a copy of the table thing:

MoonScript		Lua
	<pre>thing = { color: " red " name: " fast " width: 123 } thing_copy = { k , v for k , v in pairs thing }</pre>	
Table comprehensions, like list comprehensions, also support multiple for example we use a when clause to prevent the value associated with the c_1

MoonScript		Lua
	<pre>no_color = { k ' v for k ' v in pairs thing when k != " color " }</pre>	

The $\begin{tabular}{ll} \hline \end{tabular}$ operator is also supported. Here we create a square root look up to

MoonScript		Lua
MoonScript	<pre>numbers = { 1 / 2 / 3 / 4 } sqrts = { i / math.sqrt i for i i i </pre>	Lua
	* numbers }	

The key-value tuple in a table comprehension can also come from a single the expression should return two values. The first is used as the key and t value:

In this example we convert an array of pairs to a table where the first item the second is the value.

MoonScript		Lua
	tuples =	
	{{	
	hello "	
	/ 11	
	world	
	}	
	' { "	
	foo "	
	1	

Slicing

A special syntax is provided to restrict the items that are iterated over whe This is equivalent to setting the iteration bounds and a step size in a for I

Here we can set the minimum and maximum bounds, taking all items with inclusive:

MoonScript		Lua
MoonScript	<pre>slice = [item for item in * items [1 '5]]</pre>	Lua

Any of the slice arguments can be left off to use a sensible default. In this left off it defaults to the length of the table. This will take everything but the

MoonScript		Lua
	<pre>slice = [item for item in * items [2 ']]</pre>	

If the minimum bound is left out, it defaults to 1. Here we only provide a st bounds blank. This takes all odd indexed items: (1, 3, 5, ...)

MoonScript		Lua
	<pre>slice = [item for item in</pre>	
	items [

2]] String Interpolation

You can mix expressions into string literals using *#*{} syntax.

MoonScript		Lua
	<pre>print " I am #{ math.random ! * 100 } % sure. "</pre>	

String interpolation is only available in double quoted strings.

There are two for loop forms, just like in Lua. A numeric one and a generic

MoonScript		Lua
	<pre>for i = 10 /20 print i for k = 1 / 15 /2 an optional step pro print k for key</pre>	ovided
	, value in pairs object print key , value	

The slicing and \bullet operators can be used, just like with comprehensions:

MoonScript		Lua
	<pre>for item in * items [2 ' 4] print item</pre>	

MoonScript		Lua
	<pre>for item in * items do print item for j = 1 / 10 / 3 do print j</pre>	

A shorter syntax is also available for all variations when the body is only a

A for loop can also be used as an expression. The last statement in the bc

coerced into an expression and appended to an accumulating array table.

Doubling every even number:

MoonScript		Lua
	doubled_evens	
	= for	
	1 , 20	
	if i	
	% 2	
	== 0	
	i *	
	2 else	
	i	

You can also filter values by combining the for loop expression with the cc

For loops at the end of a function body are not accumulated into a table fo function will return nil). Either an explicit return statement can be used, (converted into a list comprehension.

MoonScript		Lua
MoonScript	<pre>func_a = -> for i = 1 , 10 do i func_b = -> return for i</pre>	Lua
	= 1	
	, 10 do i print	

func_a
!
-- prints nil
print
func_b
!
-- prints table object

This is done to avoid the needless creation of tables for functions that don

of the loop.

While Loop

The while loop also comes in two variations:

Like for loops, the while loop can also be used an expression. Additionally accumulated value of a while loop, the statement must be explicitly returned to the statement must be explicitly returned to

Continue

A continue statement can be used to skip the current iteration in a loop.

MoonScript		Lua
	i = 0 while i < 10 continue if i % 2 == 0 print i	

continue can also be used with loop expressions to prevent that iteration fr result. This examples filters the array table into just even numbers:

MoonScript		Lua
	<pre>my_numbers = { 1 / 2 / 3 / 4 / 5 / 6 } odds = for x in * my_numbers continue if x</pre>	
	% 2 == 1	
	х	

Conditionals

```
MoonScript
                                                      Lua
                       have_coins
                       =
                       false
                       if
                       have_coins
                       print
                       'n.
                       Got coins
                       н
                       else
                       print
                       'n.
                       No coins
                       н
```

A short syntax for single statements can also be used:

MoonScript		Lua
	<pre>have_coins = false if have_coins then print "</pre>	
	Got coins " else print " No coins	



Because if statements can be used as expressions, this can also be writte

MoonScript		Lua
MoonScript	<pre>have_coins = false print if have_coins then " Got coins " else " No coins "</pre>	Lua

Conditionals can also be used in return statements and assignments:

MoonScript		Lua
	is_tall =	
	(name	
) -> if	

name == н Rob н true else false message = if is_tall н Rob н н I am very tall н else н I am not so tall н print message -- prints: I am very tall

The opposite of if is unless:

MoonScript Lua unless os.date ("





With Assignment

if and elseif blocks can take an assignment in place of a conditional ex the conditional, the assignment will take place and the value that was assi conditional expression. The assigned variable is only in scope for the body meaning it is never available if the value is not truthy.

MoonScript

Lua

if user = database . find_user moon print user . name

MoonScript		Lua
	if hello = os.getenv " hello " print "	
	You have hello " hello elseif world = os.getenv " world	

"
print
"
you have world
"
world
else
print
"
nothing :(

For convenience, the for loop and if statement can be applied to single staline:

MoonScript		Lua
	print	
	II.	
	hello world "	
	if	
	name	
	== "	
	Rob	

And with basic loops:

MoonScript		Lua
	print " item: "	
	/ item for item in *	
	ltems	

Switch

The switch statement is shorthand for writing a series of if statements that value. Note that the value is only evaluated once. Like if statements, switc to handle no matches. Comparison is done with the == operator.

MoonScript		Lua
	name =	
	н	
	Dan "	
	switch	
	when	
	Robert "	
	print "	
	You are Robert "	
	when "	
	Dan "	
	/ 11	
	Daniel "	
	print "	
	Your name, it's Dan "	
	else print "	
	I don't know about your na "	ame

A switch when clause can match against multiple values by listing them ou

Switches can be used as expressions as well, here we can assign the res variable:

MoonScript		Lua
	<pre>b = 1 next_number = switch b when 1 2 when 2 3 else error "</pre>	
	" "	

We can use the then keyword to write a switch's when block on a single lir needed to write the else block on a single line.

MoonScript		Lua
	<pre>msg = switch math.random (1 , 5) when 1 then</pre>	
	you are lucky " when 2	
	then "	
	you are almost lucky "	
	else "	
	not so lucky "	

It is worth noting the order of the case comparison expression. The case's hand side. This can be useful if the case's expression wants to overwrite k by defining an eq metamethod.

In these examples, the generated Lua code may appear overwhelming. It meaning of the MoonScript code at first, then look into the Lua code if you implementation details.

A simple class:

MoonScript		Lua
	<pre>class Inventory new: => @items = {} add_item: (name) => if @items [name] #= 1 else @items [name] = 1</pre>	

A class is declared with a class statement followed by a table-like declara methods and properties are listed.
The new property is special in that it will become the constructor.

Notice how all the methods in the class use the fat arrow function syntax. instance, the instance itself is sent in as the first argument. The fat arrow I self argument.

The *e* prefix on a variable name is shorthand for *self. e* tems becomes

Creating an instance of the class is done by calling the name of the class

MoonScript		Lua
	<pre>inv = Inventory ! inv \ add_item " t-shirt " inv \ add_item " pants "</pre>	

Because the instance of the class needs to be sent to the methods when 1 operator is used.

All properties of a class are shared among the instances. This is fine for fu of objects, undesired results may occur.

Consider the example below, the clothes property is shared amongst all ir it in one instance will show up in another:

```
MoonScript
                        class
                        Person
                        clothes:
                        {}
                        give_item:
                        (
                        name
                        )
                        =>
                        table.insert
                        @clothes
                        1
                        name
                        а
                        =
                        Person
                        1
                        b
                        =
                        Person
                        1
                        а
                        \mathbf{N}
                        give_item
                        pants
                        'n.
                        b
                        \mathbf{X}
                        give_item
                        shirt
                        н
                        -- will print both pants and shirt
                        print
                        item
                        for
                        item
                        in
                        *
                        а
                        ÷
```

The proper way to avoid this problem is to create the mutable state of the

MoonScript			
	class Person new: => @clothes = {}		

Inheritance

The extends keyword can be used in a class declaration to inherit the prop another class.

MoonScript		Lua
	<pre>class BackPack extends Inventory size: 10 add_item: (name) => if # @items > size then error "</pre>	
	backpack is full	

" super name

Here we extend our Inventory class, and limit the amount of items it can ca

In this example, we don't define a constructor on the subclass, so the pare called when we make a new instance. If we did define a constructor then v method to call the parent constructor.

Whenever a class inherits from another, it sends a message to the parent _____inherited on the parent class if it exists. The function receives two argum inherited and the child class.

MoonScript	
	class
	@inherited
	;
	(child
)
	@name
	/ 11
	was inherited by "
	, child
	•

```
___name
-- will print: Shelf was inherited by Cupboard
class
Cupboard
extends
Shelf
```

Super

super is a special keyword that can be used in two different ways: It can b can be called like a function. It only has special functionality when inside ϵ

When called as a function, it will call the function of the same name in the self will automatically be passed as the first argument. (As seen in the in

When super is used as a normal value, it is a reference to the parent class

It can be accessed like any of object in order to retrieve values in the pare been shadowed by the child class.

When the \calling operator is used with super, self is inserted as the fir value of super itself. When using to retrieve a function, the raw function

A few examples of using super in different ways:

MoonScript	Lua
	class MyClass extends ParentClass a_method: => the following have the same effect: super " hello

н / 11 world 11 super \mathbf{N} a_method 11 hello н / 11 world н. super . a_method self **/** hello н / 11 world н – -- super as a value is equal to the parent clas assert super == ParentClass

super can also be used on left side of a Function Stub. The only major diff resulting function being bound to the value of super, it is bound to self.

Types

Every instance of a class carries its type with it. This is stored in the speciproperty holds the class object. The class object is what we call to build a index the class object to retrieve class methods and properties.



Class Objects

The class object is what we create when we use a class statement. The c variable of the same name of the class.

The class object can be called like a function in order to create new instan instances of classes in the examples above.

A class is made up of two tables. The class table itself, and the *base* table metatable for all the instances. All properties listed in the class declaratior

The class object's metatable reads properties from the base if they don't ϵ means we can access functions and properties directly from the class.

It is important to note that assigning to the class object does not assign int

valid way to add new methods to instances. Instead the base must explicing the base field below.

The class object has a couple special properties:

The name of the class as when it was declared is stored as a string in the object.



The base object is stored in <u>base</u>. We can modify this table to add functi have already been created and ones that are yet to be created.

If the class extends from anything, the parent class object is stored in parent

Class Variables

We can create variables directly in the class object instead of in the *base* | the property name in a class declaration.



In expressions, we can use @@ to access a value that is stored in the $__{c1}$ is shorthand for $__{self._class.hello}$.

MoonScript	
	class
	Cauptor
	Counter
	@count
	0
	new:
	=>
	@@count
	+=
	1
	Counter
	Counter
	Councer
	!
	print
	Counter
	•
	count
	prints 2

The calling semantics of *ee* are similar to *e*. Calling a *ee* name will pass

argument using Lua's colon syntax.

MoonScript		Lua
	@@bollo	
	1	
	2	
	' 3	
	' 4	

Class Declaration Statements

In the body of a class declaration, we can have normal expressions in adc this context, self is equal to the class object.

Here is an alternative way to create a class variable compared to what's d

MoonScript		Lua
	class Things @class_var = " hello world "	

These expressions are executed after all the properties have been added

All variables declared in the body of the class are local to the classes prop for placing private values or helper functions that only the class methods c

MoonScript	
	class MoreThings
	secret
	=
	log
	=
	(msa
)
	->
	LOG:

```
msg
some_method:
=>
log
"
hello world:
"
...
secret
```



When and are prefixed in front of a name they represent, respective self and self._class.

If they are used all by themselves, they are aliases for self and self._clas

MoonScript		Lua
	<pre>assert @ == self assert @ == selfclass</pre>	

For example, a quick way to create a new instance of the same class from @

MoonScript		Lua
	<pre>some_instance_method = () => @@</pre>	

Class Expressions

The class syntax can also be used as an expression which can be assign returned.

MoonScript	
	Х
	=
	<pre>class Bucket drops: 0 add_drop: => @drops += 1</pre>
	\bot

Anonymous classes

The name can be left out when declaring a class. The <u>name</u> attribute will expression is in an assignment. The name on the left hand side of the ass <u>nil</u>.

MoonScript	
	BigBucket
	_ class
	extends
	Bucket
	=>
	@drops
	+=
	assert
	Bucket
	н
	BigBucket

You can even leave off the body, meaning you can write a blank anonymo

X =	MoonScript		
× =			
		X	
		=	
CLASS		class	

Export Statement

Because all assignments to variables that are not lexically visible will be d syntax is required to declare a variable globally.

The export keyword makes it so any following assignments to the specifie assigned locally.

MoonScript		Lua
	<pre>export var_name 'var_name2 var_name3 " hello " '" world "</pre>	

This is especially useful when declaring what will be externally visible in a

MoonScript		Lua
	my_module.moon module " my_module " '	
	export print_result	

length = (Х , у) -> math.sqrt X * Х + у * У print_result = (Х 1 y) -> print ii. Length is н , length Х 1 У -- main.moon require н my_module шĺ my_module . print_result 4 **'**5 -- prints the result print my_module . length 6



Assignment can be combined with the export keyword to assign to global

MoonScript		Lua	
	export some_number		
	, message_str = 100		
	/ 11		
	hello world "		

Additionally, a class declaration can be prefixed with the export keyword ir

Export will have no effect if there is already a local variable of the same na

Export All & Export Proper

The export statement can also take special symbols * and \land .

export * will cause any name declared after the statement to be exported

export ^ will export all proper names, names that begin with a capital lette

Local Statement

Sometimes you want to declare a variable name before the first time you a statement can be used to do that.

In this example we declare the variable a in the outer scope so its value (if statement. If there was no local statement then a would only be access statement.

MoonScript		Lua
	<pre>local a if something a = 1 print a</pre>	

local can also be used to shadow existing variables for the rest of a scop

MoonScript		Lua
	x = 10 if something local x x =	



When you have one function that calls another, you typically order them si can access the first. If both functions happen to call each other, then you r names:

MoonScript		Lua
	<pre>local first ' second first = -> second ! second = -> first !</pre>	

The same problem occurs with declaring classes and regular values too.

Because forward declaring is often better than manually ordering your ass local is provided: MoonScript Lua local * first = -> print data second 1 second = -> first 1 data = {}

local * will forward declare all names below it in the current scope.

Similarly to export one more special form is provided, <u>local</u>. This will for begin with a capital letter.

Often you want to bring some values from a table into the current scope a name. The import statement lets us accomplish this:

MoonScript		Lua
	import insert from table	

The multiple names can be given, each separated by a comma:

MoonScript		Lua
	<pre>import C , Ct , Cmt from lpeg</pre>	

Sometimes a function requires that the table be sent in as the first argume syntax). As a shortcut, we can prefix the name with a n to bind it to that the table be sent in a shortcut, we can be a shortcut to be a shortcut.

MoonScript		Lua
	<pre> some object my_module = state: 100 add: (value) => self . state + value import value import add from my_module print add 22 equivalent to calling r</pre>	my_module\get 22

When handing multiple imports you can substitute the comma with a newl whitespace. When working with a lot of imports you might write something

MoonScript	Lua
<pre>import assert_csrf assert_timezo not_found require_login from require " helpers "</pre>	one 1

With Statement

A common pattern involving the creation of an object is calling a series of series of properties immediately after creating it.

This results in repeating the name of the object multiple times in code, add common solution to this is to pass a table in as an argument which contain values to overwrite. The downside to this is that the constructor of this object

The with block helps to alleviate this. Within a with block we can use a s with either \Box or ∇ which represent those operations applied to the object

MoonScript Lua with Person T name = н **Oswald** 11 Υ. add_relative my_dad Υ. save L print name

For example, we work with a newly created object:

The with statement can also be used as an expression which returns the access to.

MoonScript		Lua
	file =	
	with	
	File "	
	favorite_foods.txt "	
	X	
	set_encoding "	
	utf8	

Or...

MoonScript		Lua
	create_person =	
	(

name 1 relatives) -> with Person 1 . name = name \mathbf{N} add_relative relative for relative in * relatives me = create_person н. Leaf н. ' { dad 1 mother 1 sister }

In this usage, with can be seen as a special form of the K combinator.

The expression in the $\ensuremath{\mbox{with}}$ statement can also be an assignment, if you w expression.

MoonScript		Lua
	with str Hello print priginal: str priginal: upper:	
When used as a statement, do works just like it does in Lua.

MoonScript		Lua
	do	
	var	
	=	
	н	
	hello	
	п	
	print	
	var	
	print	
	var	
	- nil bere	
	IIII HELE	

MoonScript's d_0 can also be used an expression . Allowing you to combin The result of the d_0 expression is the last statement in its body.

MoonScript		Lua
	<pre>counter = do i = 0 -> i += 1 i print counter</pre>	



MoonScript Lua tbl = { key: do print " assigning key! 1234 }

Destructuring assignment is a way to quickly extract values from a table b array based tables.

Typically when you see a table literal, [1,2,3], it is on the right hand side c is a value. Destructuring assignment swaps the role of the table literal, and side of an assign statement.

This is best explained with examples. Here is how you would unpack the f

thing =	MoonScript		Lua
<pre>{ 1 ' 2 } { a ' b } = thing print a ' b</pre>		<pre>thing = { 1 , 2 } { a , b } = thing print a , b</pre>	

In the destructuring table literal, the key represents the key to read from the value represents the name the read value will be assigned to.

MoonScript		Lua
	<pre>obj = { hello: " world " day: " tuesday " length: 20 } { hello: bello</pre>	
	/day: the_day } = obj print hello /the_day	

This also works with nested data structures as well:

MoonScript Lua obj2 = { numbers: { 1 <mark>′</mark>2 **'** 3 , 4 } properties: { color: н green " height: 13.5 } } { numbers: { first 1 second }} = obj2 print first 1 second 1 color

If the destructuring statement is complicated, feel free to spread it out ove complicated example:

MoonScript Lua { numbers: { first 1 second } properties: { color: color } } = obj2

It's common to extract values from at table and assign them the local varian name as the key. In order to avoid repetition we can use the : prefix oper

MoonScript		Lua
	{ :concat ':insert } =	
	table	

This is effectively the same as import, but we can rename fields we want t syntax:

MoonScript

{
 :mix
 'imax
 'random:
 rand
 }
 =
 math

Destructuring In Other Places

Destructuring can also show up in places where an assignment implicitly t this is a for loop:



hello " **/** world } { " egg " / 11 head " } } for { left / right } in * tuples print left , right

We know each element in the array table is a two item tuple, so we can ur names clause of the for statement using a destructure.

Function Stubs

It is common to pass a function from an object around as a value, for exar method into a function as a callback. If the function expects the object it is argument then you must somehow bundle that object with the function so

The function stub syntax is a shorthand for creating a new closure function object and function. This new function calls the wrapped function in the co

Its syntax is the same as calling an instance method with the $\hfill operator \hfill provided.$

MoonScript		Lua
	<pre>my_object = { value: 1000 write: => print " the value: " value value value value value run_callback = (func) -> print " running callback" " func</pre>	

```
-- this will not work:
-- the function has to no reference to my_objec
run_callback
my_object
.
write
-- function stub syntax
-- lets us bundle the object into a new function
run_callback
my_object
\
write
```

The Using Clause; Controlling Destructive Assignme

While lexical scoping can be a great help in reducing the complexity of the get unwieldy as the code size increases. Consider the following snippet:

MoonScript		Lua
	<pre>i = 100 many lines of code my_func = -> i = 10 while i > 0 print i i -= 1 my_func ! print i will print 0</pre>	

In my_func , we've overwritten the value of i mistakenly. In this example it consider a large, or foreign code base where it isn't clear what names hav

It would be helpful to say which variables from the enclosing scope we interprevent us from changing others by accident.

The using keyword lets us do that. Using nil makes sure that no closed va assignment. The Using clause is placed after the argument list in a functio are no arguments.

MoonScript		Lua
	<pre>i = 100 my_func = (using nil) -> i = " hello " a new local variable is my_func ! print i prints 100, i is unaffe</pre>	s created here

Multiple names can be separated by commas. Closure values can still be modified:

MoonScript	Lua

tmp = 1213 i , k = 100 , 50 my_func = (add using k ' i) -> tmp = tmp + add -- a new local tmp is created i += tmp k += tmp my_func (22) print i 1 k -- these have been updated

Misc.

Implicit Returns on Files

By default, a file will also implicitly return like a function. This is useful for v can put your module's table as the last statement in the file so it is returne

Writing Modules

Lua 5.2 has removed the module function for creating modules. It is recoministead when defining a module.

We can cleanly define modules by using the shorthand hash table key/val

```
MoonScript
                                                      Lua
                       MY_CONSTANT
                       =
                       н
                       hello
                       н
                       my_function
                       =
                       ->
                       print
                       н
                       the function
                       my_second_function
                       =
                       ->
                       print
                       another function
                       ш
                       {
                       :my_function
                       1
```

:my_second_function
':MY_CONSTANT
}

If you need to forward declare your values so you can access them regard you can add local * to the top of your file.

License (MIT)

Copyright (C) 2015 by Leaf Corcoran

Permission is hereby granted, free of charge, to any person obt of this software and associated documentation files (the "Softw in the Software without restriction, including without limitati to use, copy, modify, merge, publish, distribute, sublicense, a copies of the Software, and to permit persons to whom the Softw furnished to do so, subject to the following conditions: The above copyright notice and this permission notice shall be all copies or substantial portions of the Software. THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHAN FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES C LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER I THE SOFTWARE.

Generated on Sun Sep 25 20:08:51 2016; MoonScript v0.5.0