

# CORE LANGUAGE

## Comments

# Comment to end of line.  
#-...-# Multi-line comment.

## Identifier

A string start with an underscore or letter, followed by some underscore, letters or numbers (case sensitive). Identifiers are generally used as names of objects or variables.

## Reserved Identifiers

if	elif	else	while	for	def
end	class	break	continue	return	true
false	nil	var	do	import	as

## Operators

( )	[ ]	.	-	!	~	*	/	%	+	-	<<
>>	&	^		..	<	<=	>	>=	==	!=	&&
	?	:	=	+=	-=	*=	/=	%=	&=	=	^=
<<=	>>=	{	}								

## String

'...' "..."

string delimiters; special characters need to be escaped:

\a	bell	\b	backspace	\f	form feed
\n	newline	\r	return	\t	tab
\v	vert. tab	\\	backslash	\'	single quote
\"	double quote	\?	question	\0	NULL
\ooo	character represented octal number.				
\xhh	character represented hexadecimal number.				

## Types

nil	Means no value (written as <code>nil</code> ).
boolean	Contains <code>true</code> and <code>false</code> .
integer	Signed integer number.
real	Floating point number.
string	Can include any character (and zero).
function	First class type, can be assigned as a value.
class	Instance template, read only.
instance	Object constructed by class.
module	Read-write key-value pair table.
list	Variable-length ordered container class.
map	Read-write hash key-value container class.
range	Integer range class.

## Variable and Assignment examples

a = 1	Simple assignment (or declare variables).
var a	Declare variables and initialize to <code>nil</code> .
var a, b	Declare multiple variables.
var a=0, b=1	Declare multiple variables and initialize.
a = 1 + 3	Operation and assignment.

## Expression and Statement

**expression** Consist of operators, operands, and grouping symbols (brackets), etc. All expressions are evaluable.

**statement** The most basic execution unit. Consists of an assignment expression or function call expression.

Statement examples:

4.5	A simple expression, just an operand.
!true	Logical not expression, unary operation.
1+2	An addition expression, binary operation.
print(12)	Function call expression.

## Operators in precedence order

() (call)	[] (index)	.(field)
!	~	-(negative)
*	/	%
+	-	
<<	>>	(bitwise shift operators)
&		(bitwise and)
^		(bitwise xor)
		(bitwise or)
..		(connect or range)
<	<=	>
>	>=	
==	!=	
&&		(stops on <code>false</code> , returns last evaluated value)
		(stops on <code>true</code> , returns last evaluated value)
+	-	
? :		(conditional expression)
=		(= and other assignment operators)

## Conditional expression

*condition ? expression1 : expression2*

If the value of *condition* is `true`, then *expression1* will be executed, otherwise *expression2* will be executed. The conditional expression return the the last evaluated value.

## Logical operations and Boolean

The condition detection operation require a Boolean value, and non-boolean type will do the following conversion:

nil	Convert to <code>false</code> .
number	0 is converted to <code>false</code> , others are converted to <code>true</code> .
instance	Try to use the result of the <code>tobool()</code> method, otherwise it will be converted to <code>true</code> .
other	Convert to <code>true</code> .

## Scope, blocks and chunks

**block** Is the body of a control structure, body of a function or a chunk. The block consists of several statements.

**chunk** A file or string of script.

Variables defined in the chunk have a global scope, and those defined in other blocks have a local scope.

## Control structures

**if** *cond block* {**elif** *cond block*} [**else block**] **end**

**do** *block* **end**

**while** *cond block* **end**

**for** *id : expr block* **end** iterative statement.

**for** *id = expr, cond[, expr] block* **end** loop for statement, (not support now).

**break** exits loop (must be in **while** or **for** statement).

**continue** start the next iteration of the loop (must be in **while** or **for** statement).

**return** [*expr*] exit function and return a (`nil`) value.

NOTE: **expression** aka. **expr**; **identifier** aka. **id**; and **condition** aka. **cond**.

## Function and Lambda expression

**def** *name (args) block* **end**

A named function is a statement, the *name* is a identifier.

**def** (*args*) *block* **end**

An anonymous function is an expression.

**/args-> expr**

Lambda expression, the return value is *expr*.

**id {, id}**

Arguments list (aka. **args**), Lambda expression arguments list can omit “,”.

## Class and Instance

**class name [: super]**

{**var id**{, **id**} | **def id (args) block end**}

**end**

class consists of the declaration of some member variables and methods. **name** is the class name (an identifier); **super** is the super class (an expression).

## List Instance

**l=[]** New empty list value.

**l=[0]** The list has a value “0”.

**l=[[ ],nil]** **l[0]==[ ]** and **l[1]==nil**; different types of values can be stored in the list.

## Map Instance

**m={}** New empty map value.

**m=[0:'ok', 'k':nil]** **l[0]=='ok'** and **l['k']==nil**; the key can be any value that is not nil.

## Range Instance

**r=0..5** New range from 0 to 5.

## Exception handling

**throw exception [, message]**

Throw a **exception** value and unnecessary **message** value.

**try**

**block {**

**except ((expr {, expr} | ..) [as id [, id]] | ..)**

**block**

**} end**

One or more **except** blocks must exist. Only runtime exceptions can be catch.

Some **except** statements examples:

**except ..** Catch all exceptions, but no exception variables.

**except 0,1 as ..** Capture 0 and 1, no exception variables.

**except .. as e** Capture all exception to variable **e**.

**except 0 as e** Capture exception 0 to variable **e**.

**except .. as e,m** Capture all exception to variable **e**, and save the message to variable **m**.

**classname(object)**

Get the class name of **object**. The **object** is a class or an instance.

**classof(object)**

Get the class of **object**, and return nil when it fails.

**number(expr) int(expr) real(expr)**

Convert **expr** to a number (automatically detect integer or real), integer or real respectively, and return 0 or 0.0 if the conversion fails.

**str(expr)**

Convert **expr** to a string. For instance, it will try to call the **tostring** method.

**module([name])**

Create an empty module, and name is an optional module name.

**size(expr)**

Get the length of the string or instance (by calling the **size** method).

**compile(text [, mode])**

When **mode** is 'string', **text** is evaluated as a script, and when **mode** is 'file', a script file whose path is **text** is read and evaluated. The mode is 'string' by default.

**issubclass(sub, sup)**

Returns **true** if **sub** (class) is **sup** (class or instance) or its derived class, otherwise return **false**.

**isinstance(obj, base)**

Returns **true** if **obj** is an instance of **base** (class or instance) or its derived class, otherwise return **false**.

**open(path[, mode])**

Open a file by **path** and return an instance of this file. The file is opened in the specified **mode**:

'r' read-only mode, the file must exist.

'w' write-only mode, always create a empty file.

'a' Create a empty file or append to the end of an existing file.

'r+' read-write mode, the file must exist.

'w+' read-write mode, always create a empty file.

'a+' read-write mode, create a empty file or append to the end of an existing file.

'b' binary mode, it can be combined with other access modes.

## File Members

**file.write(text)**

Write the **text** to the file.

**file.read([count])**

If the **count** is specified, the number of bytes will be read, otherwise the entire file will be read.

**file.readline()**

Read a line from the file (the newline character is determined by the platform).

**file.seek(offset)**

Set the file pointer to **offset**.

**file.tell()**

Get the offset of the file pointer.

**file.size()**

Get the size of the file.

**file.flush()**

Flush the file buffer.

# BASIC LIBRARY

## Global Functions

**assert(expr [, msg])**

Throw 'assert\_failed' when **expr** is **false**, and **msg** is an optional exception message.

**print(...)**

Print all arguments to stdout.

**input([prompt])**

Read a line of text from stdin, **prompt** is optional prompt message.

**super(object)**

Get the super class of **object**. The **object** is a class or an instance.

**type(expr)**

Get the type name string of **expr**.

```
file.close()
```

Close the file.

## List Members

```
list.init(args)
```

Constructor, put the elements in *args* into list one by one.

```
list.toString()
```

Serialized the list instance.

```
list.push(value)
```

Append the *value* to the tail of the list.

```
list.pop([index])
```

Remove the element at *index* (the default index is  $-1$ ) from the list.

```
list.insert(index, value)
```

Insert the *value* before the element at *index*.

```
list.item(index)
```

Get the element at *index*. The *index* can be an integer, and a list or range instance.

```
list.setitem(index, value)
```

Set the element referenced at *index* to *value*.

```
list.size()
```

Get the number of elements in the list instance.

```
list.resize(expr)
```

Modify the number of elements to the value of *expr*. The added elements are set to `nil`, and the reduced elements are discarded.

```
list.clear()
```

Clear all elements in the list instance.

```
list.iter()
```

Get the iterator function of the list instance.

```
list.concat()
```

Serialize and concatenate all elements in the list instance into a string.

```
list.reverse()
```

Reverse the order of all elements in the list instance.

```
list.copy()
```

Copy the list instance, not copy the element but keep the reference.

```
list() .. expr
```

Append the value of *expr* to the tail of the list instance and return that instance.

```
list() + list()
```

Concatenate two list instances and return the left operand instance.

```
list() == expr
```

Check if two list instances are equal. It checks all elements one by one.

```
list() != expr
```

Check if two list instances are not equal. It checks all elements one by one.

## Map Members

```
map.init()
```

Constructor.

```
map.toString()
```

Serialized the map instance.

```
map.insert(key, value)
```

Insert a key-value pair and return `true`, and return `false` when the insertion fails (e.g. the pair already exists).

```
map.remove(key)
```

Remove the key-value pair by the *key*.

```
map.item(key)
```

Get the value mapped by the *key*. It will throw a "key\_error" exception when the key-value pair does not exist.

```
map.setitem(key, value)
```

Set the *value* mapped by the *key*. If the key-value pair does not exist, a new one will be inserted.

```
map.find(key)
```

Get the value mapped by the *key*. It will return `nil` when the key-value pair does not exist.

```
map.size()
```

Get the number of key-value pairs in the map instance.

```
map.iter()
```

Get the value iterator function of the map instance.

## Range Members

```
rang.init(lower, upper)
```

The constructor. The range is from *lower* to *upper*, and the step is 1.

```
rang.toString()
```

Serialized the rang instance.

```
rang.lower()
```

Get the *lower* value of the range instance.

```
rang.upper()
```

Get the *upper* value of the range instance.

```
rang.iter()
```

Get the value iterator function of the range instance.

# THE STRING LIBRARY

## Import Module

```
import string
```

## Basic operations

```
string.count(s, sub[, begin[, end]])
```

Count the number of occurrences of the *sub* string in the string *s*. Search from the position between *begin* and *end* of *s* (default is 0 and `size(s)`).

```
string.split(s, pos)
```

Split the string *s* into two substrings at position *pos*, and returns the list of those strings.

```
string.split(s, sep[, num])
```

Splits the string *s* into substrings wherever *sep* occurs, and returns the list of those strings. Split at most *num* times (default is `string.count(s, sep)`).

```
string.find(s, sub[, begin[, end]])
```

Check whether the string *s* contains the substring *sub*. If the *begin* and *end* (default is 0 and `size(s)`) are specified, they will be searched in this range.

```
hex(number)
```

Convert *number* to hexadecimal string.

```
byte(s)
```

Get the code value of the first byte of the string *s*.

```
char(number)
```

Convert the *number* used as the code to a character.

## Formatting

```
string.format(fmt[, args])
```

Returns a formatted string. The pattern starting with '%' in the formatting template *fmt* will be replaced by the value of [*args*]: %[flags][fieldwidth][.precision]type

## Types

<code>%d</code>	Decimal integer.
<code>%o</code>	Octal integer.
<code>%x</code> <code>%X</code>	Hexadecimal integer lowercase, uppercase.
<code>%x</code> <code>%X</code>	Octal integer.
<code>%f</code>	Floating-point in the form <code>[-]nnnn.nnnn</code> .
<code>%e</code> <code>%E</code>	Floating-point in exp. form <code>[-]n.nnnn e [+ -]nnn</code> , uppercase if <code>%E</code> .
<code>%g</code> <code>%G</code>	Floating-point as <code>%f</code> if $-4 < \text{exp.} \leq \text{precision}$ , else as <code>%e</code> ; uppercase if <code>%G</code> .
<code>%c</code>	Character having the code passed as integer.
<code>%s</code>	String with no embedded zeros.
<code>%q</code>	String between double quotes, with special characters escaped.
<code>%%</code>	The <code>'%'</code> character (escaped).

## Flags

<code>-</code>	Left-justifies, default is right-justify.
<code>+</code>	Prepends sign (applies to numbers).
<code>(space)</code>	Prepends sign if negative, else space.
<code>#</code>	Adds <code>"0x"</code> before <code>%x</code> , force decimal point; for <code>%e</code> , <code>%f</code> , leaves trailing zeros for <code>%g</code> .

## Field width and precision

<code>n</code>	Puts at least <code>n</code> characters, pad with blanks.
<code>0n</code>	Puts at least <code>n</code> characters, left-pad with zeros.
<code>.n</code>	Use at least <code>n</code> digits for integers, rounds to <code>n</code> decimals for floating-point or no more than <code>n</code> chars. for strings.