Core Language

Full documentation

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`
`try`	`except`	`raise`	`static`		

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 `nil`).
boolean	Contains `true` and `false`.
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 `nil`.
`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.
walrus	Combines an assignment to a variable which value can be used as an expression.

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.
`print(a := 12)`	Walrus assigment and expression.

Operators in precedence order

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

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 `false`.
number	0 converted to `false`.
string	Empty string converted to `false`.
bytes	Empty bytes buffer converted to `false`.
comptr	0 (`NULL`) converted to `false`.
comobj	0 (`NULL`) converted to `false`.
instance	Try to use the result of the `tobool()` method, otherwise it will be converted to `true`.
other	Convert to `true`.

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.

`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**.

Modules

Berry has some predefined modules (like `math`). You can extend the runtime with your own modules, either as Berry code or native code.

`import` ***name*** `[as` ***variable***`]`
Load the module ***name*** and store in local or global variable ***name*** or ***variable*** if latter is defined.

Once a module is loaded, you can't change its content unless you use module `import introspect`, see below.

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***}}
| {`static var` ***id***{`=`***expr***}}{`,` ***id***{`=`***expr***}}
| [`static`] **`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).

Members and static members

Methods have an implicit first argument `self` used to access members.
Static methods (or Class methods) have an implicit `_class` argument to access the `class` object.

Accessing members

instance`.`***key***
Access the instance method or variable by literal name.

instance`.(`***string***`)`
Access the instance method or variable dynamically by string.
Use ***instance***`.a` or ***instance***`.("a")`

class`.`***key***
Access the class (static) method or variable by literal name.

class`.(`***string***`)`
Access the class (static) method or variable dynamically by string.

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` included.

Exception handling

`throw` ***exception*** [`,` ***message***]
Throw a ***exception*** value and optional ***message*** value.

`try`
block {
`except` ((***expr*** {`,` ***expr***} | `..`) [**`as`** ***id*** [`,` ***id***]] | `..`)
block
} **`end`**
One or more **`except`** blocks must exist. Only runtime exceptions can be caught.

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*.

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***.

`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.

`bool(`***expr***`)`
Convert ***expr*** to a bool.

`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`.

`call(`***function***`[,` ***args*** `][,` ***list*** `])`
Call a ***function*** with arbitrary number of arguments, all ***args*** are pushed as static arguments. If the last argument is a ***list***, all elements are pushed as elementary arguments.

`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(`***string*** | ***bytes***`)`
Write the ***text*** or ***raw bytes*** 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`.readbytes(`[***count***]`)`
Return raw bytes instead of string. 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.

file`.close()`
Close the file.

List Members

Full documentation

`list()` or `list(`***args***`)`
Constructor, put the elements in ***args*** into list one by one. Also use `[]`.

list`[`***index***`]`
Can be used to read or write at ***index***, raises an exception if index is out of bounds. Equivalent to `list.item()` and `list.setitem()`.

list`[`***a*** `..` ***b***`]`
Returns a sub-list containing elements from index ***a*** to ***b*** included. If ***b*** is omitted it includes all elements to the end of thelist. If ***b*** is negative, it counts from the end of the list (ex `list[1 .. -2]` removes the first and last elements). Equivalent to `list.item()`.

list`t[`***list***`]`
Returns a sub-list from the indices of the list, returns `nil` element if an index is out of bounds. Equivalent to `list.item()`.

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, raises an exception if index is out of bounds.

list`.setitem(`***index***`,` ***value***`)`
Set the element referenced at ***index*** to ***value***, raises an exception if index is out of bounds.

list`.size()`
Get the number of elements in the list instance; equivalent of `size(`***list***`)`.

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`.keys()`
Return a `range` object containing indices of the list.

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

Full documentation

`map()`
Constructor. Also use `{}`.
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`.insert(`***key***`,` ***value***`)`
Set the ***value*** mapped by the ***key*** only if ***key*** does not exist, and returns `true`. Returns `false` and do not update the value if ***key*** already exists.
map`.contains(`***key***`)`
Returns `true` if the map contains the ***key***.
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; equivalent of `size(`***map***`)`.
map`.iter()`
Get the iterator function over the values of the map instance.
map`.keys()`
Get the iterator function over the keys of the map instance.

Range Members

Full documentation

`range(`***lower***`,` ***upper***{, ***increment***}`)`
The constructor. The range is from ***lower*** to ***upper***, and the step is 1 or ***increment***. ***increment*** can be negative.
range`.tostring()`
Serialized the rang instance.
range`.iter()`
Get the value iterator function of the range instance.
range`.lower()`
Get the ***lower*** value of the range instance.
range`.upper()`
Get the ***upper*** value of the range instance.
range`.incr()`
Get the ***increment*** value of the range instance.
range`.setrange(`***lower***`,` ***upper***{, ***increment***}`)`
Changes the ***lower***, ***upper***, and ***increment***, does not change an existing iterator.

The String Library

`import string`
Full documentation

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. Returns -1 if not found.
`string.startswith(`***s***`,` ***sub***[`,` ***case_insensitive***]`)`
`string.endswith(`***s***`,` ***sub***[`,` ***case_insensitive***]`)`
Check whether the string ***s*** starts/ends with the substring ***sub***; case-insensitive if ***case_insensitive*** is `true`.
`string.hex(`***number***`)`
Convert ***number*** to hexadecimal string.
`string.byte(`***s***`)`
Get the code value of the first byte of the string ***s***.
`string.char(`***number***`)`
Convert the ***number*** used as the code to a character.

Transformation

`string.toupper(`***text***`)`
`string.tolower(`***text***`)`
Convert the ***text*** to uppercase or lowercase; ASCII only no support for Unicode.
`string.tr(`***text***`,` ***chars***`,` ***char_or_empty_string***`)`
Replaces in ***text*** any occurrence of character(s) from ***chars*** to a single character, or remove if empty string.
`string.replace(`***text***`,` ***text1***`,` ***text2***`)`
Replaces in ***text*** occurrence of ***text1*** with ***text2*** (this is slower than `string.tr()`
`string.escape(`***text***`[,` ***berry_mode***`] )`
Escapes the string with double quotes suitable for C, if ***berry_mode*** is `true` escape to single quotes suitable for Berry.

Formatting

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

`%d`	`%i`	Decimal integer.
`%u`		Unsigned decimal integer.
`%o`		Octal integer.
`%x`	`%X`	Hexadecimal integer lowercase, uppercase.
`%f`		Floating-point in the form [-]nnnn.nnnn.
`%e`	`%E`	Floating-point in exp. form [-]n.nnnn e [+|-]nnn, uppercase if `%E`.
`%g`	`%G`	Floating-point as `%f` if $-4 <$ exp. $\leq$ precision, else as `%e`; uppercase if `%G`.
`%c`		Character having the code passed as integer.
`%s`		String.
`%q`		Escaped string.
`%%`		The `'%'` character (escaped).

Flags

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

Field width and precision

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

Simplified Formatting with f-strings

An alternative syntax using f-strings allows more compact formatting. They are synctactic sugar around `format()` function, so they have the same performance.

f-strings are preceded by `f` and can use single or double quotes. String can be split on several literals and lines.

```
f"This uses double quotes"
f'This uses single quotes'
f"This" 'uses' "a combination" 'of quotes'
```

Values and expressions are surrounded by `{ }`

```
f"Hello {name}"
f"1 + 1 is {1 + 1}"
```

For brackets, use double-brackets. JSON example:

```
f'{{"name":"{name}"}}'
```

The default format is `%s` (string). You can specify a format after a colon ':'. The character '%' is not required.

```
f"The price is {price:.2g}"
```

For fast debugging, use equal sign '=' to dump a value with its name:

```
f"{name=} {price=:.2g}"
# format("name=%s price=%.2g", name, price")
# name=bob price=12.34
```

More examples:

'`f"a = {self.a}"`' is '`format("a = %s", self.a)`'
'`f"{self.a:04i}"`' is '`format("%04i", self.a)`'
'`f"{self.a=}"`' is '`format("self.a=%s", self.a)`'
'`f"{self.a=:g}"`' is '`format("self.a=%g", self.a)`'

The Math Library

`import math`
Full documentation

Constants

`math.pi`
Pi number (`3.14159` or `3.141592654` depending or resolution).

`math.nan`
`NaN` Not-a-Number used to indicate an invalid number. Fun fact: `math.nan != math.nan`

`math.imin`
Smallest possible integer depending on compilation options (`-2147483648` or `-9223372036854775808`).

`math.imax`
Biggest possible integer depending on compilation options (`2147483647` or `9223372036854775807`).

Integer conversion

`math.floor(`***value***`)`
Return the rounded down ***value*** as `real`.

`math.ceil(`***value***`)`
Return the rounded up ***value*** as `real`.

General functions

`math.abs(`***value***`)`
Return the positive absolute value of ***value*** as `real`.

`math.rand()`
Return a random `int`. This is not cryptographic quality.

`math.srand(`***int***`)`
Seed the random generator with ***int***.

`math.isnan(`***value***`)`
Return `true` if ***value*** is a `NaN` Not-a-Number.

Log & Exponent

`math.sqrt(`***value***`)`
Return the square root of ***value***.

`math.log(`***value***`)`
Return the natural logarithm of ***value***.

`math.log10(`***value***`)`
Return the logarithm in base 10 of ***value***.

`math.exp(`***value***`)`
Return the natural exponent of ***value***.

`math.pow(`***x***`, `***y***`)`
Return ***x*** to the power of ***y***.

Trigonometry

`math.sin(`***value***`)`
`math.cos(`***value***`)`
`math.tan(`***value***`)`
Return the sine, cosine, tangent of ***value*** (`int` or `real`) in radians, returns a `real`.

`math.asin(`***value***`)`
`math.acos(`***value***`)`
`math.atan(`***value***`)`
Return the arc sine, arc cosine, arc tangent of ***value*** (`int` or `real`) in radians, returns a `real`.

`math.atan2(`***y***`, `***x***`)`
Return the arc tangent of ***y*** `/` ***x*** in radians, works even if ***x*** is zero.

`math.deg(`***value***`)`
Convert radians to degrees.

`math.rad(`***value***`)`
Convert degrees to radians.

Hyperbolic

`math.sinh(`***value***`)`
`math.cosh(`***value***`)`
`math.tanh(`***value***`)`
Return the hyperbolic sine, cosine, tangent of ***value*** (`int` or `real`) in radians, returns a `real`.

The Bytes Library

`bytes()` is a native class used to manipulate raw bytes.
Full documentation

`bytes()`
Constructor for an empty `bytes` object.

`bytes(`***size***`)`
Constructor for an empty `bytes` object, pre-allocate ***size*** bytes to optimize memory allocation.
If ***size*** is negative, pre-allocate (`-`***size***) bytes and make object fixed size, filling with zeros.

`bytes(`***comptr***`, `***size***`)`
Constructor to a `bytes` object mapped at a fixed memory location ***comptr*** and of fixed ***size***.

General Functions

bytes`.size()`
Return the size of content in bytes; equivalent of `size(`***bytes***`)`.

`bytes.resize(`***size***`)`
Resize the object to ***size*** bytes, truncate or fill with zeros if needed; unless the buffer is fixed size.

bytes`.clear()`
Reset the object to an empty `bytes()`; unless the buffer is fixed size.

bytes`.reverse([`***start***`, [`***len***`, [`***grouplen***`]]])`
Reverse the bytes from ***start*** over ***len*** (or full buffer if not specified) over groups of ***grouplen*** bytes (or single bytes). This is useful for RGB pixel manipulation.

bytes`.copy()`
Copy to a new separate object.

bytes`==`***bytes***
bytes`!=`***bytes***
Return `true` if content of ***bytes*** are equal or different.

bytes `..` ***bytes***
Append the second ***bytes*** to the first ***bytes***.

bytes `+` ***bytes***
Create a new ***bytes*** buffer containing the concatenation of both ***bytes***.

bytes`.ismapped()`
Return `true` if the buffer is mapped to a fixed location in memory.

Accessor Functions

bytes`[`***index***`]`
Read or write byte at ***index*** as `int`; throws an exception if index is out of bounds.

bytes`[`***start***`..`***end***`]`
Return a new instance of `bytes` containing bytes from ***start*** to ***end*** included. Indices can be out of bounds. If ***end*** is omitted, copy to the end of the buffer. If ***start*** or ***end*** are negative, count from end of buffer (`-1` is last byte).

bytes`.get(`***offset***`, `***size***`)`
Read the value at ***offset*** as an unsigned integer of ***size*** bytes (***size*** can be 1, 2, 3, 4 for Little Endian or −2, −3, −4 for Big Endian). Return `0` if indices are out of bounds.

bytes`.geti(`***offset***`, `***size***`)`
Same as `get` above as signed integer.

bytes`.set(`***offset***`, `***value***`[, `***size***`])`
Set the value at ***offset*** as an unsigned integer of ***size*** bytes (default 1) with ***value*** (***size*** can be 1, 2, 3, 4 for Little Endian or −2, −3, −4 for Big Endian). No effect if indices are out of bounds.

bytes`.seti(`***offset***`, `***value***`[, `***size***`])`
Same as `set` above as signed integer.

bytes`.add(`***value***`, `***size***`)`
Append ***value*** to the ***bytes*** buffer as ***size*** bytes (***size*** can be 1, 2, 3, 4 for Little Endian or −2, −3, −4 for Big Endian).

bytes`.getfloat(`***offset***`[, `***big_endian***`])`
Read the value at ***offset*** as a 4 bytes floating point number. If ***big_endian*** is `true` read as Big Endian.

bytes`.setfloat(`***offset***`, `***value***`[, `***big_endian***`])`
Set the value at ***offset*** to a 4 bytes floating point number from ***value***. If ***big_endian*** is `true` read as Big Endian.

bytes`.getbits(`***offset_bits***`, `***len_bits***`, `***value***`)`
Read at bit level from ***offset_bits*** of ***len_bits***.

bytes`.setbits(`***offset_bits***`, `***len_bits***`)`
Set at bit level from ***offset_bits*** of ***len_bits*** with ***value***.

bytes`.setbytes(`***offset***`, `***bytes2***`, [`***start***`, [`***len***`]])`
Set buffer at ***offset*** from ***bytes2***; copy entier buffer or only from ***start*** with ***len***.

Conversion Functions

bytes`.tostring([`***max_size***`])`
Convert ***bytes*** buffer to a string representation. To prevent memory exhaustion, only 32 bytes or to ***max_size***.

bytes`.tohex()`
Convert ***bytes*** buffer to a hex `string`, without `bytes()` decorator.

bytes`.fromhex(`***string***`)`
Replace ***bytes*** buffer from ***string*** as hex string.

bytes`.asstring()`
Convert ***bytes*** buffer to a `string` containing the raw bytes.

bytes`.fromstring(`***string***`)`
Replace ***bytes*** buffer from ***string*** as raw bytes.

bytes`.tob64()`
Convert ***bytes*** buffer to a base64 `string`.

bytes`.fromb64(`***string***`)`
Replace ***bytes*** buffer from ***string*** as base64.

The Global Library

`import global`
Full documentation

`global()`
Return the list of all global variables.

`global.contains(`***id***`)`
Return `true` if the global variables exists.

`global.member(`***id***`)`
`global.`***id***
Return value of global variable ***id*** or `nil` if it does not exists.

`global.(`***string***`)`
Return value of global variable ***string*** by name or `nil` if it does not exists. Example: `global.("a")`

`global.setmember(`***id***`,` ***value***`)`
`global.`***id*** `=` ***value***
Set global variable ***id*** to ***value***, create the global variable if needed.

The JSON Library

Full documentation

`json.load(`***string***`)`
Concatenate ***string*** into a complete path.

`json.dump(`***any***`)`
Convert ***any*** to a JSON string.

The Instrospect Library

`import introspect`

Full documentation

`introspect.members(`***any***`)`
Return the list of names of members for the `class`, `instance` or `module`.

`introspect.members()`
Return the list of global variables, equivalent to `global()`.

`introspect.get(`***any***`,` ***id***`)`
Read the attribute ***id*** for ***any***, returns `nil` if key does not exist.

`introspect.set(`***any***`,` ***id***`,` ***value***`)`
Set the attribute ***id*** for ***any*** to ***value***.

`introspect.name(`***any***`)`
Return the name of `any` (function, class or module) or ***nil***.

`introspect.ismethod(`***function***`)`
Return `true` if the ***function*** is a method of a class, `false` it it's a standalone function.

`introspect.module(`***name***`)`
Import module passed by ***name***.

`introspect.setmodule(`***name***`,` ***any***`)`
Change the value for module ***name***; use with caution as it can disrupt the runtime.

`introspect.toptr(`***int***`)` `introspect.fromptr(`***comptr***`)`
Convert an ***int*** to ***comptr*** and backwards, works only for platforms where integers and pointers are the same size.

The OS Library

Full documentation

`os.getcmd()`
Get the path of the current directory.

`os.chdir(`***path***`)`
Switch the current folder to the ***path***.

`os.mkdir(`***path***`)`
Create a level of directory (with ***path***).

`os.remove(`***path***`)`
Delete file of directory form ***path***.

`os.listdir([`***path***`])`
Return a list of file and folder names contained in the specified ***path*** (the default is `'.'`).

`os.system(`***cmd***`[,` ***args***`])`
Execute a system command.

`os.exit()`
Exit the interpreter process.

The `os.path` Module

`os.path.isdir(`***path***`)`
Check if the ***path*** is a folder.

`os.path.isfile(`***path***`)`
Check if the ***path*** is a file.

`os.path.exists(`***path***`)`
Check if the ***path*** already exists.

`os.path.split(`***path***`)`
Split the ***path*** into dir-name and base-name.

`os.path.splitext(`***path***`)`
Split the ***path*** into file-name and ext-name.

`os.path.splitext(`***args***`)`
Concatenate ***args*** into a complete path.