

Contents

1. Introduction	03
2. Skeleton of Aegisub Scripts	03
3. Aegi	04
3.1. ffm	04
3.2. mff	04
3.3. progressTitle	05
3.4. progressTask	05
3.5. progressSet	06
3.6. progressReset	06
3.7. progressCancelled	07
3.8. progressLine	07
3.9. videoIsOpen	08
3.10. getFramerate	08
4. Logger	09
4.1. Fatal	09
4.2. Error	09
4.3. Warn	09
4.4. Hint	09
4.5. Debug	09
4.6. Trace	09
4.7. Log	010
4.8. Assert	010
4.9. windowError	011
4.10. windowAssertError	011
4.11. lineWarn	012
4.12. lineError	012
5. Ass	013
5.1. Line Collection	014
5.2. Loop through collected lines	015
5.3. Remove Lines	016
5.4. Insert Line	016
5.5. Commit changes to the subtitle	017
5.6. Create Lines	018
5.7. Add Style	018
5.8. Update Metadata	018
5.9. Parse Line	018
5.10. Get selection	018
6. Table	019
6.1. isEmpty	019
6.2. isList	020
6.3. shallowcopy	020
6.4. deepcopy	021
6.5. map	021
6.6. filter	022
6.7. size	023
6.8. contains	024
6.9. keys	025

6.10. count	025
6.11. pop	026
6.12. reverse	027
6.13. slice	027
6.14. extend	028
6.15. uniq	029
6.16. diff	029
6.17. intersect	030
6.18. removeIndices	031
6.19. removeValues	031
6.20. makeSet	032
6.21. prepend	033
6.22. splice	033
6.23. invert	035
6.24. merge	035
6.25. equal	036
6.26. view	036

1. Introduction

ILL is a module that aims to make working with subtitle objects efficient. It does most of the heavy lifting so that we can do more with fewer lines of code in our script. It provides various functions that allow us to work with lines, tags, drawings, text and comments. This makes it unnecessary to reinvent the wheel and write your own functions to do most of the common task while still allowing you to do complex tasks more easily.

2. Skeleton of Aegisub Scripts

```
export script_name = "name of the script"
export script_description = "description of your script"
export script_version = "0.0.1"
export script_author = "you"
export script_namespace = "namespace of your script"

DependencyControl = require "l0.DependencyControl"
depctrl = DependencyControl{
  {
    {
      "ILL.ILL"
      version: "0.0.1"
      url: "https://github.com/TypesettingTools/ILL-Aegisub-Scripts"
      feed: "https://raw.githubusercontent.com/TypesettingTools/ILL-Aegisub-Scripts/main/DependencyControl.json"
    }
  }
}
ILL = depctrl\requireModules!
{:Aegi, :Ass} = ILL

functionName = (sub, sel, act) ->
  -- stuff goes here

depctrl\registerMacro functionName
```

This is the framework that all your scripts will have. Here we import ILL as well as classes that ILL offers that we need in this script. Here, we only import Aegi and Ass class but ILL offers much more and user should import them as they are needed. Finally, we define a function called functionName. This function name is what we register in Aegisub in the last line, and it gets executed as soon as we run the script. Everything we do in the guide below will go inside the function where --stuff goes here is written.

3. Aegi

Class Aegi deals with various Aegisub api and provides convenience function wrappers around them.

3.1. ffm

ffm uses the loaded framerate data to convert absolute time in milliseconds to a frame number.

3.1.1. Arguments

Argument	Description	Type	Default
ms	Time in milliseconds	Integer	-

3.1.2. Returns

If video is not open, it returns nil.

Returns	Description	Type
frame or nil	Frame corresponding to time	integer or nil

3.1.3. Usage

```
frame = Aegi.ffm 1000  
frame = Aegi.ffm line.start_time
```

3.2. mff

mff uses loaded framerate data to convert a frame number of the video into an absolute time in milliseconds.

3.2.1. Arguments

Argument	Description	Type	Default
frame	Frame number of video	integer	-

3.2.2. Returns

If video is not open, it returns nil.

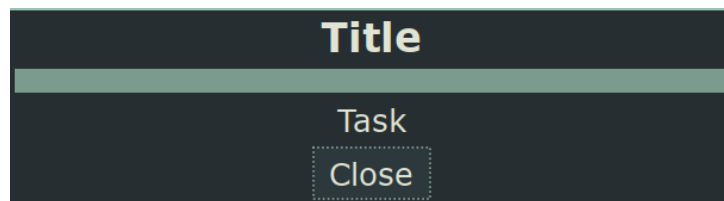
Returns	Description	Type
millisecond or nil	Time in milliseconds	integer or nil

3.2.3. Usage

```
ms = Aegi.mff 100  
line.start_time = Aegi.mff 500
```

3.3. progressTitle

A progress dialog box is always shown when an Aegisub script is run. Aegisub allows user to modify what is shown in it. `progressTitle` sets the title for the progress window. Title is the large text displayed above the progress bar. This text should usually not change while the script is running. By default, title is set to the name of the macro running.



3.3.1. Arguments

Argument	Description	Type	Default
title	Title to set int the progress window	String	-

3.3.2. Returns

Returns
Nil

3.3.3. Usage

```
Aegi.progressTitle "Title of the progress window."
```

3.4. progressTask

`progressTask` sets the text for the progress window. Task is the small text below the progress bar showing what the script is currently doing. It is updated by the user as the task of the script changes.

3.4.1. Arguments

Argument	Description	Type	Default
task	Task that the script is currently executing	String	""

3.4.2. Returns

Returns
Nil

3.4.3. Usage

```
Aegi.progressTask "Plotting World Domination."
```

3.5. progressSet

progressSet generates a progress bar in the progress window ranging from 0% to 100%.

3.5.1. Arguments

Argument	Description	Type	Default
i	Current index	Integer	-
n	Total index	Integer	-

3.5.2. Returns

Returns
Nil

3.5.3. Usage

```
for i = 1, 20  
  Aegi.progressSet i, 20
```

3.6. progressReset

progressReset resets all the progress by setting the progress bar to 0% and progress task to empty string.

3.6.1. Arguments and Returns

Argument	Returns
Nil	Nil

3.6.2. Usage

```
Aegi.progressReset!
```

3.7. progressCancelled

progressCancelled checks if the user has cancelled the execution of script. In such a case, it stops any further execution of the script. It should be used inside loops or callback functions such that the user does not have wait until it is completed to stop the execution of the script.

3.7.1. Arguments and Returns

Argument	Returns
Nil	Nil

3.7.2. Usage

```
Aegi.progressCancelled!
```

3.8. progressLine

progressLine is a combination of a couple of functions that we have seen above so that we can execute all of them at once. It is recommended to use this unless for some reason you need to execute them separately.

This sets the progress bar, sets the progress task and checks for user cancellation. This is ideal to be used inside a loop or callback function.

3.8.1. Arguments

Argument	Description	Type	Default
Line	Line table as given by Ass class	Table	-
i	Current index	Integer	-
n	Total index	Integer	-

More info about getting line table from Ass class is discussed later.

3.8.2. Returns

Returns
Nil

3.8.3. Usage

```
ass = Ass sub, sel
ass\iterLines (l, i, n) ->
    Aegi.progressLine l, i, n
```

3.9. videoIsOpen

videoIsOpen is used to find if video is currently open in Aegisub or not.

3.9.1. Arguments

Argument
Nil

3.9.2. Returns

Returns	Description	Type
videoState	State of video	True if open and false if closed

3.9.3. Usage

```
if Aegi.videoIsOpen!  
  -- video is open  
else  
  -- video is not open
```

3.10. getFramerate

getFramerate returns the framerate of the video that is currently open. If the video is not open, it returns the default value of 24000 / 1001.

3.10.1. Arguments

Argument
Nil

3.10.2. Returns

Returns	Description	Type
framerate	Framerate of the video	Float

3.10.3. Usage

```
framerate = Aegi.getFramerate!
```


4. Logger

Class `Logger` allows you to log messages to the progress window. If messages are logged by a script, the progress window stays open after the script has finished running until the user clicks the `Close` button.

Logger has two components: Log level and messages. Log level indicates the severity level of the message while messages are the string that you want to log.

By default, Aegisub's log level is set to 3 which means that the message above 3 won't be seen by end user unless they set the log level higher themselves.

4.1. Fatal

These message indicate something really bad happened and the script cannot continue. The log level of these messages is 0. Level 0 messages are always shown regardless of trace level settings in Aegisub. The execution of the script will end after this message is logged.

4.2. Error

An error indicates the user should expect something to have gone wrong even though you tried to recover. A fatal error might happen later. The log level of error messages is 1.

4.3. Warn

A warning indicates something is wrong and the user ought to know because it might mean something needs to be fixed. The log level of warning messages is 2.

4.4. Hint

A hint indicates something that the user should know that is not necessarily a cause for alarm. The log level of hint messages is 3.

4.5. Debug

A debug message includes information meant to help fix errors in the script, such as dumps of variable contents. Since the default trace level of Aegisub is 3, debug messages, which has the log level of 4, won't be seen by average user of the script unless they manually changed it. This is useful for script writers during the debugging of their scripts.

4.6. Trace

A track message includes extremely verbose information about what the script is doing, such as a message for each single step done with lots of variable dumps. The log level of trace message is 5.

4.7. Log

This is simply a wrapper around hint message who don't want to use hint and want to use log which seems synonymous to simply logging messages.

Arguments

Argument	Description	Type	Default
messages	Message you want to show	string / number / boolean / table	-
...	Parameters to the format string	lua string.format parameters	-

Returns

Returns
Nil

4.7.1. Usage

```
Logger.fatal "This is a fatal message." -- log level 0
Logger.error "This is an error message." -- log level 1
Logger.warn "This is a warning message." -- log level 2
Logger.hint "This is a hint message." -- log level 3
Logger.debug "This is a debug message." -- log level 4
Logger.trace "This is a trace message." -- log level 5

Logger.log "This is a simple message." -- log level 3
```

You can also use Lua's string.format method.

```
Logger.fatal "There are %d %s messages.", 5, "fatal"
```

This will yield the message: There are 5 fatal messages.

4.8. Assert

assert will only show a message if a condition is false. After showing the message, further execution of script is terminated.

4.8.1. Argument

Argument	Description	Type	Default
condition	Condition to check	lua statement or boolean	-
message	Message you want to show if condition is false	string / number / boolean / table	-

Argument	Description	Type	Default
...	Parameters to the format string	lua string.format parameters	-

4.8.2. Returns

Returns
Nil

4.8.3. Usage

```
Logger.assert 1 > 2, "Unfortunately, 1 was not greater than 2."
```

4.9. windowError

Instead of using progress window, if the user wants to use Aegisub dialog to show messages, this can be used. Currently this only supports text messages. The execution of the script will be terminated after the message is shown.

4.9.1. Arguments

Argument	Description	Type	Default
message	Message you want to show	string / number / boolean / table	-
...	Parameters to the format string	lua string.format parameters	-

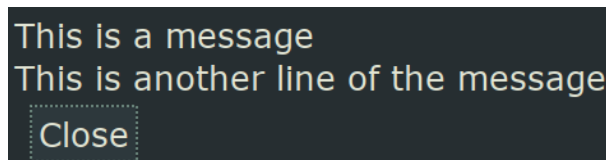
4.9.2. Returns

Returns
Nil

4.9.3. Usage

```
Logger.windowError "This is a message\nThis is another line of the message"
```

The above command will yield:



4.10. windowAssertError

windowAssertError is the same as the [Section 4.9](#) windowError and [Section 4.8](#) assert combined.

4.10.1. Arguments and Returns

Same as [Section 4.8](#)

4.10.2. Usage

```
Logger.windowAssertError line.start_time < line.end_time, "This is a message\nThis  
is another line of the message"
```

4.11. lineWarn

lineWarn is used for showing warning for a particular line.

4.11.1. Arguments

Argument	Description	Type	Default
line	line collected by Ass class	table	-
message	Message you want to show	string	"not specified"

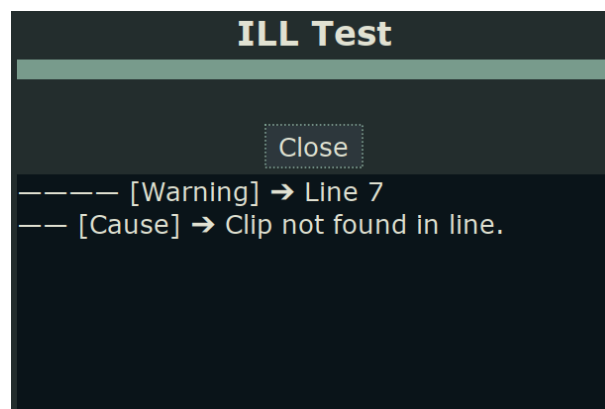
4.11.2. Returns

Returns
Nil

4.11.3. Usage

```
ass = Ass sub, sel  
ass\iterLines (l, i, n) ->  
  Logger.lineWarn l, "Clip not found in line."
```

The code above yields:



4.12. lineError

lineError is used for showing critical warning for a particular line. The execution of the script is terminated after showing this message.

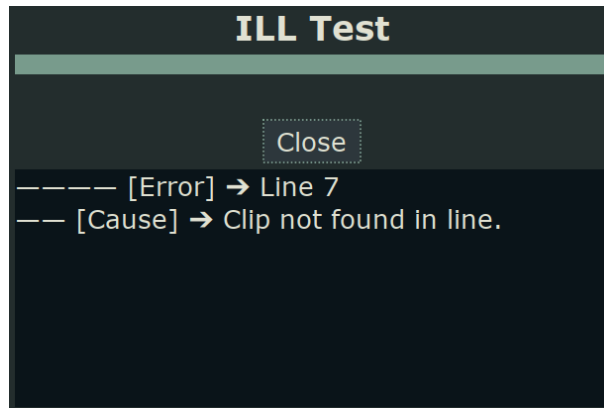
4.12.1. Arguments and returns

Same as [Section 4.11](#)

4.12.2. Usage

```
ass = Ass sub, sel
ass\iterLines (l, i, n) ->
    Logger.lineError l, "Clip not found in line."
```

The code above yields:



5. Ass

One of the first things we do when we write a script is we initiate the Ass instance. It's job is to collect the basic information of the subtitle files, lines of the files and provide various methods to act on them.

```
main = (sub, sel, act) ->
    ass = Ass sub, sel, act
```

Arguments

Argument	Description	Type	Default
sub	Subtitle object given by Aegisub	userdata	-
sel	(Optional) Selected lines	table[integer]	-
act	(Optional) Active line	integer	-
callback	(Optional) condition for line collection	function	(l) -> not l.comment

5.1. Line Collection

As we initialize the Ass class, we are collecting lines to work on at the same time. By default, the condition (line) -> line.comment suggests that only uncommented lines are collected but that can be changed.

The collected line table will have lines with all the [fields of a normal line table](#) but it adds other fields.

Argument	Description	Type
duration	duration of line in milliseconds	integer
startFrame	start frame of a line	integer
endFrame	end frame of a line	integer
frameCount	total number of frames	integer
styleRef	style table of current line	table
absoluteIndex	actual index of line in subtitle file	integer
naturalIndex	index of line as seen in Aegisub	integer

5.1.1. Collect all dialogue lines

If you only send sub as an argument, all uncommented lines will be collected.

Note

All lines does not mean all lines of the subtitle. It means all the dialogue lines as seen in Aegisub.

```
main = (sub, sel, act) ->
  ass = Ass sub
```

5.1.2. Collect all selected lines

If you send both sub and sel as arguments, only the uncommented dialogue lines among the selected lines will be collected.

```
main = (sub, sel, act) ->
  ass = Ass sub, sel
```

5.1.3. Choosing conditions for collection

There may be cases where you want to only collect certain types of lines. In that case, you can override the default condition for line collection.

If you don't like that it skips over commented lines and want it to be included, collect lines as shown below. It will select all selected lines without checking anything.

```
ass = Ass sub, sel, act, -> return true
```

In fact, you can give any condition here and if that condition is fulfilled, only those lines will be collected i.e. if the callback function returns `true`, the line will be collected and vice versa. For example, if you want to only collect commented lines.

```
ass = Ass sub, sel, act, -> return line.comment
```

If you want to only collect lines whose layer is 1:

```
ass = Ass sub, sel, act, ->
    return line.layer == 1
```

If you want to only collect lines whose text contain certain substring:

```
ass = Ass sub, sel, act, ->
    return line.text\match "text"
```

5.2. Loop through collected lines

Now that we have collected lines, we might want to iterate through it and work on individual lines.

```
main = (sub, sel, act) ->
    ass = Ass sub, sel
    return if #ass.lines == 0          -- Return early if no lines were collected.
    ass\iterLines (l, i, n) ->
        -- do things to individual line here.
```

5.2.1. Arguments

One of the argument of `ass\iterLines` is a callback function which means it will offer you a few arguments that you can use.

Argument	Description	Type	Default
<code>l</code>	line table	table	-
<code>i</code>	current iteration of line	integer	-
<code>n</code>	total number of lines in collection	integer	-

The argument `l` is the line table which contains all the information about current line. However, the other arguments `i` and `n` are purely for progress reporting. An example was shown in [Section 3.8](#).

5.2.2. Returns

While iterating through the lines, if we want to stop the iteration, we can return `false`.

Returns	Description	Type
stop iteration	if iteration should be stopped	boolean

For example:

```

main = (sub, sel, act) ->
  ass = Ass sub, sel
  return if #ass.lines == 0
  ass\iterLines (l, i, n) ->
    if condition
      return false
      -- As soon as this callback function returns false, the iteration
will stop.

```

5.3. Remove Lines

If you want to remove lines from the collection, you can mark them for removal during the iteration.

Warning

This only marks the line for removal. Line has not been actually removed from the subtitle yet. Look [Section 5.5](#) for more.

5.3.1. Arguments

Argument	Description	Type	Default
line	line collected by Ass class	table	-

5.3.2. Returns

Returns
Nil

5.3.3. Usage

```

main = (sub, sel, act) ->
  ass = Ass sub, sel
  return if #ass.lines == 0
  ass\iterLines (l, i, n) ->
    if i % 3 == 0
      ass\removeLine l

```

5.4. Insert Line

You want to insert lines to the subtitle at some point. If you are iterating through the collection, make a copy of the line before inserting it or you can create a line from scratch to insert at any time. Look [Section 5.6](#) for how to create lines from scratch.

Warning

This only marks the line for insertion. Line has not been actually inserted to the subtitle yet. Look [Section 5.5](#) for more.

5.4.1. Arguments

Argument	Description	Type	Default
line	line collected by Ass class	table	-
index	(Optional) index of the collection at which to insert the line	integer	#ass.lines

In case the user gives the index too low or high, the index is clamped to be 1 and total number of lines so that it is always within the collection. If index is not provided, it will add the line at the end of the collection.

5.4.2. Returns

Returns
Nil

5.4.3. Usage

```
main = (sub, sel, act) ->
  ass = Ass sub, sel
  return if #ass.lines == 0
  ass\iterLines (l, i, n) ->
    line = Table.deepcopy l
    -- Make some changes to the copy
    line.layer += 1
    line.text = "This is a copy of line #{line.naturalIndex}"
    if i % 3 == 0
      ass\insertLine line -- inserts line at the end of collection
    elseif i % 5 == 0
      ass\insertLine line, i -- inserts line right after current line
```

5.5. Commit changes to the subtitle

After we make various changes to the line collection, this is where we actually commit the changes to the subtitle file. This removes all the lines that were marked for removal, inserts lines at the index the user asked them to insert and apply the changes the user makes on the line.

5.5.1. Arguments

Argument	Description	Type	Default
updateRefs	update references of the line after comitting	boolean	false

Note

When the user commits and makes changes to the subtitle file, the line collection will become outdated. The indices may point to wrong lines. If the user changes times of the line, the fields related to frames also become wrong. If the user does not want to work with the line collection again (i.e. the script no longer has to iterate through lines again), there is no need to update references. However, if the user wants to iterate through lines again, we need to fix the references so that we're sure we're working with proper lines.

5.5.2. Returns

Returns
Nil

5.5.3. Usage

```
main = (sub, sel, act) ->
  ass = Ass sub, sel
  return if #ass.lines == 0
  ass\iterLines (l, i, n) ->
    if i % 3 == 0
      line = Table.deepcopy l
      line.text = "This is a copy of line #{line.naturalIndex}"
      ass\insertLine line, i
    elsif i % 5 == 0
      ass\removeLine l
  ass\commit!
```

```
ass\commit true -- if you want to update the line collection
```

5.6. Create Lines

5.7. Add Style

5.8. Update Metadata

5.9. Parse Line

5.10. Get selection

This gets the table of selected lines. This is most useful to be returned by the main function of the script and whatever indices are present in the table, those lines will be selected after the script is run.

5.10.1. Arguments

Argument
Nil

5.10.2. Returns

Returns	Description	Type
sel	indices of selected lines	table

5.10.3. Usage

```
main = (sub, sel, act) ->
  ass = Ass sub, sel
  return if #ass.lines == 0
  ass\iterLines (l, i, n) ->
    -- make changes here
  ass\commit!
  return Ass\getSelection!
```

As the user removes line or inserts line, the selection table gets updated. Therefore, `getSelection` returns the proper indices after working with lines.

6. Table

Table consists of various actions user can perform on the Lua tables. Lua does not differentiate between a dictionary and a list. Therefore, user must first verify if the function below is applicable to the table before applying. The function will be tagged with `List` for arrays, `Table` for dictionary and will be tagged with both if the function can be used for either.

6.1. isEmpty

Tags: List, Table

`isEmpty` is used to find if the table is empty.

6.1.1. Arguments

Argument	Description	Type	Default
table	table to check if it is empty	table	-

6.1.2. Returns

Returns	Description	Type
table empty state	true if table is empty, false otherwise	boolean

6.1.3. Usage

```
if Table.isEmpty {1, 2, 3}
  -- table is empty
else
  -- table is not empty
```

6.2. isList

Tags: List, Table

isList is used to find if the table is a list or a dictionary.

6.2.1. Arguments

Argument	Description	Type	Default
table	table to check if it is list or dictionary	table	-

6.2.2. Returns

Returns	Description	Type
table list state	true if table is list, false otherwise	boolean

6.2.3. Usage

```
if Table.isList {1, 2, 3}
  -- table is list
else
  -- table is dictionary
```

6.3. shallowcopy

Tags: List, Table

shallowcopy as the name suggests makes a shallow copy of the table

6.3.1. Arguments

Argument	Description	Type	Default
table	table to check if it is list or dictionary	table	-

6.3.2. Returns

Returns	Description	Type
shallow copy	shallow copy of the table	boolean

6.3.3. Usage

```
copy = Table.shallowcopy {1, 2, 3}
```

6.4. deepcopy

Everything about deepcopy is same as [Section 6.3](#): shallowcopy above except it returns a deep copy of the table.

6.5. map

Tags: List, Table

map is used to apply a user defined function to each element of the table.

6.5.1. Arguments

Argument	Description	Type	Default
table	table to map elements of	table	-
callback function	function to apply to each element	function	-

6.5.1.1. Arguments of callback function

Argument	Description	Type	Default
value	value that you are modifying	any except nil/NaN	-
key	key in case of table, index in case of list	any except nil/NaN	-

6.5.1.2. Returns of callback function

Returns	Description	Type
value	modified value	any except nil/NaN

6.5.2. Returns

Returns	Description	Type
modified table	table where each element is returned by callback function	table

6.5.3. Usage

```
tbl = {1, 2, 5}
tbl = Table.map tbl, (value) -> value + 1

-- Output
-- tbl = {2, 3, 6}
```

```
tbl = {1, 2, 5, "apple"}
tbl = Table.map tbl, (value) ->
  if type(value) == "string"
    value = value\gsub "p", "b"
  else
    value *= 2
  return value

-- Output
-- tbl = {2, 4, 10, "abble"}
```

```
tbl = {
  "key1": 1
  "key2": 2
  "key3": 5
}
tbl = Table.map tbl, (v, k) ->
  if k == "key2"
    return v
  else
    return v + 1

-- Output
-- tbl = {
--   "key1": 2
--   "key2": 2
--   "key3": 6
-- }
```

6.6. filter

Tags: List, Table

filter is used to only keep elements which satisfy a condition.

6.6.1. Arguments

Argument	Description	Type	Default
table	table to filter elements of	table	-
callback function	function that defines the condition	function	-

6.6.1.1. Arguments of callback function

Argument	Description	Type	Default
value	value that you are modifying	any except nil/NaN	-
key	key in case of table, index in case of list	any except nil/NaN	-

6.6.1.2. Returns of callback function

Returns	Description	Type
state	condition has passed or not	boolean

6.6.2. Returns

Returns	Description	Type
modified table	table where each element is returned by callback function	table

6.6.3. Usage

```
tbl = { 1, 2, 3, 4, 5, 6, 7, 8, 9}
tbl = Table.filter tbl, (v) -> v % 3 == 0

-- Output
-- tbl = {3, 6, 9}
```

```
tbl = {
  "key1": "foo"
  "key2": "bar"
  "key3": 5
  "key4": 10
}
tbl = Table.filter tbl, (v, k) ->
  if k == "key4"
    return true
  else
    return type(v) == "string"

-- Output
-- tbl = {
--   "key1": "foo"
--   "key2": "bar"
--   "key4": 10
-- }
```

6.7. size

Tags: List, Table

size is used to find the total number of elements in a table. While you can use #tbl to find the number of elements in a list, you cannot do the same for a dictionary.

For example, in a table `tbl = {1, "foo": 1, 2, "bar": 2}`, #tbl will return 2 as it will not count key value pair in the table.

6.7.1. Arguments

Argument	Description	Type	Default
table	table whose size must be calculated	table	-

6.7.2. Returns

Returns	Description	Type
size	number of elements of table	integer

6.7.3. Usage

```
tbl = { 1, "foo": 1, 2, "bar": 2}
count = Table.size tbl

-- Output
-- count = 4
```

6.8. contains

Tags: List, Table

contains is used to find if a table contains a certain value. In a list, it checks if an element exists in array. In a key-value pair, it checks if any key has that certain value.

6.8.1. Arguments

Argument	Description	Type	Default
table	table to check the value in	table	-
value	value to check	number / boolean / string	-

6.8.2. Returns

Returns	Description	Type
state	true if value exists, false otherwise	boolean

6.8.3. Usage

```
tbl = { 1, 2, 3, 4, 5, 6, 7, 8, 9}
Table.contains tbl 10 -- returns false
Table.contains tbl 7  -- returns true
```



```
tbl = { 1, 2, "foo": "bar", 3, 4}
Table.contains tbl, "bar" -- returns true
```

6.9. keys

Tags: Table, List

keys is used to get a list of keys of a table. In a list, it will return an array of indices but in a key-value pair, it will return an array of keys only.

6.9.1. Arguments

Argument	Description	Type	Default
table	table to find keys of	table	-

6.9.2. Returns

Returns	Description	Type
list	list of keys	table

6.9.3. Usage

```
tbl = {"foo": 1, "bar": 2}
keys = Table.keys tbl

-- Output
-- keys = {"foo", "bar"}
```

```
tbl = {"foo", "bar"}
keys = Table.keys tbl

-- Output
-- keys = {1, 2}
```

6.10. count

Tags: Table, List

count is used to count how many elements satisfy a condition.

6.10.1. Arguments

Argument	Description	Type	Default
list	list to count element of	table	-
callback function	function that defines the condition	function	-

6.10.1.1. Arguments of callback function

Argument	Description	Type	Default
value	value that you are modifying	any except nil/NaN	-
key	key in case of table, index in case of list	any except nil/NaN	-

6.10.1.2. Returns of callback function

Returns	Description	Type
count	number of elements that pass the condition	boolean

6.10.2. Returns

Returns	Description	Type
occurence	number of occurence of value	integer

6.10.3. Usage

```
tb = {1, 2, 3, 1, 4, 1}
result = Table.count tb, (v) -> v == 1

-- Output
-- result = 3

result = Table.count tb, (v) -> v > 2

-- Output
-- result = 2
```

6.11. pop

Tags: **List only**

pop is used to remove an element from the list.

6.11.1. Arguments

Argument	Description	Type	Default
list	list to remove element of	table	-
index	(Optional) index of element to remove	integer	last element of list

If index is not provided, the last element of the array is removed.

6.11.2. Returns

Returns	Description	Type
table	table with elements popped	table

6.11.3. Usage

```
tbl = Table.pop {1, 2, 3, 4, 5}
```

```
-- Output  
-- {1, 2, 3, 4}
```

```
tbl = Table.pop {1, 2, 3, 4, 5}, 3 -- remove third element
```

```
-- Output  
-- {1, 2, 4, 5}
```

6.12. reverse

Tags: **List only**

reverse is used to reverse elements from the list.

6.12.1. Arguments

Argument	Description	Type	Default
list	list to reverse element of	table	-

6.12.2. Returns

Returns	Description	Type
table	table with elements reversed	table

6.12.3. Usage

```
tbl = Table.reverse {1, 2, "foo", 3, 4, "bar", 5}
```

```
-- Output  
-- {5, "bar", 4, 3, "foo", 2, 1}
```

6.13. slice

Tags: **List only**

slice is used to extract a section of a list based on the specified start, end and step parameters.

6.13.1. Arguments

Argument	Description	Type	Default
list	list to extract section of	table	-
start index	(Optional) starting index of the slice	integer	1
end index	(Optional) ending index of the slice	integer	last item of the list
step	(Optional) step size for iteration	integer	1

6.13.2. Returns

Returns	Description	Type
table	extracted section of list	table

6.13.3. Usage

```
tb = {10, 20, 30, 40, 50}

result = Table.slice tb, 2, 4 -- extract from index 2 to 4

-- Output
-- result = {20, 30, 40}

result = Table.slice tb, 1, 5, 2 -- extract from index 1 to 5 with step 2

-- Output
-- result = {10, 30, 50}

result = Table.slice tb, 4, 2, -1 -- extract from index 4 to 2 with step -1 i.e.
in reverse

-- Output
-- {40, 30, 20}
```

6.14. extend

Tags: **List only**

extend is used to extend a list by appending elements from one or more lists.

6.14.1. Arguments

Argument	Description	Type	Default
list	list to extend	table	-
...	any number of lists separated by comma	table	-

6.14.2. Returns

Returns	Description	Type
table	combined lists	table

6.14.3. Usage

```
list1 = {1, 2, 3}
list2 = {4, 5}
list3 = {6, 7, 8}
result = Table.extend list1, list2, list3

-- Output
-- result = {1, 2, 3, 4, 5, 6, 7, 8}
```

6.15. uniq

Tags: **List only**

uniq is used to remove duplicate elements from a list.

6.15.1. Arguments

Argument	Description	Type	Default
list	list to count element of	table	-

6.15.2. Returns

Returns	Description	Type
occurence	number of occurence of value	integer

6.15.3. Usage

```
list = {1, 2, 3, 1, 4, 2, 5}
result = Table.uniq list

-- Output
-- result = {1, 2, 3, 4, 5}
```

6.16. diff

Tags: **List only**

diff is used to remove elements from a list that are present in any of the additional lists.

6.16.1. Arguments

Argument	Description	Type	Default
list	base list	table	-
...	any number of lists separated by comma	table	-

6.16.2. Returns

Returns	Description	Type
list	list with elements only in first list	table

6.16.3. Usage

```
list = {1, 2, 3, 4, 5, 6}
list1 = {3, 4}
list2 = {1, 2}

result = Table.diff list, list1, list2

-- Output
-- result = {5, 6}
```

6.17. intersect

Tags: **List only**

intersect is used to find elements that exists in all lists.

6.17.1. Arguments

Argument	Description	Type	Default
...	any number of lists separated by comma	table	-

6.17.2. Returns

Returns	Description	Type
list	list with elements found in all input lists	table

6.17.3. Usage

```
list = {1, 2, 3, 4, 5, 6}
list1 = {1, 3, 5}
list2 = {1, 5, 10}

result = Table.intersect list, list1, list2

-- Output
-- result = {1, 5}
```

6.18. removeIndices

Tags: **List only**

removeIndices removes the elements at the specified indices.

6.18.1. Arguments

Argument	Description	Type	Default
list	list to remove indices of	table	-
indices	indices to remove	table	-

6.18.2. Returns

Returns	Description	Type
list	list with elements of the indices removed	table

6.18.3. Usage

```
tb = {10, 20, 30, 40, 50}
result = Table.removeIndices tb, {2, 4}

-- Output
-- result = {10, 30, 50}
```

6.19. removeValues

Tags: **List only**

removeValues is used to remove elements with a specific value or list of values from a list.

6.19.1. Arguments

Argument	Description	Type	Default
list	list to remove values from	table	-
value(s)	value or list of values	number / string / boolean	-

6.19.2. Returns

Returns	Description	Type
list	list with elements with value(s) removed	table

6.19.3. Usage

```
tb = {"foo", "bar", "baz", "foo"}
result = Table.removeValues tb, "foo"
```

```
-- Output
-- result = {"bar", "baz"}
```

```
tb = {"foo", "bar", "baz", "foo"}
result = Table.removeValues tb, {"bar", "baz"}
```

```
-- Output
-- result = {"foo", "foo"}
```

6.20. makeSet

Tags: **List only**

makeSet is used make a set of a list i.e. a key-value table where key is the element of the list and value is true. The sets have no duplicates. This is mostly useful for fast lookups of unique elements present in a list.

6.20.1. Arguments

Argument	Description	Type	Default
list	list to make set of	table	-

6.20.2. Returns

Returns	Description	Type
table	table that is set of input list	table

6.20.3. Usage

```
tb = {"foo", "bar", "baz", "foo"}
result = Table.removeValues tb, "foo"
```

```
-- Output
-- result = {
--   foo: true
--   bar: true
--   baz: true
-- }
```


6.21. prepend

Tags: **List only**

prepend is used to add a single or list of elements to the beginning of the list.

6.21.1. Arguments

Argument	Description	Type	Default
list	list to prepend elements to	table	-
value(s)	value or list of values	number / string / boolean	-

6.21.2. Returns

Returns	Description	Type
list	list with elements prepended	list

6.21.3. Usage

```
tb = {3, 4, 5}
result = Table.prepend tb, {1, 2}

-- Output
-- result = {1, 2, 3, 4, 5}
```

```
tb = {3, 4, 5}
result = Table.prepend tb, "foo"

-- Output
-- result = {"foo", 3, 4, 5}
```

6.22. splice

Tags: **List only**

splice is used to modify a list by adding or replacing elements starting at a specific position while also removing a specified number of elements. This can replace, insert, remove element at any index at once.

6.22.1. Arguments

Argument	Description	Type	Default
list	list to modify	table	-
start	index at which to start adding or removing elements	integer	-
delete	number of elements to remove starting at start index	integer	-

Argument	Description	Type	Default
...	any number of elements to be inserted at start index	any	-

6.22.2. Returns

Returns	Description	Type
removes	list of removed elements	list

The original list is modified in place so it is not returned.

6.22.3. Usage

6.22.3.1. Replace Elements

```
tb = {1, 2, 3, 4, 5}
result = Table.splice tb, 2, 2, 9, 10

-- Output
-- tb = {1, 9, 10, 4, 5}
-- result = {1, 2, 3, 4, 5}
```

1. Starts at index 2
2. Removes 2 elements {2, 3}
3. Inserts {9, 10} at index 2

6.22.3.2. Insert Elements Only

```
tb = {1, 2, 3}
result = Table.splice tb, 2, 0, 8, 9

-- Output
-- tb = {1, 8, 9, 2, 3}
-- result = {}
```

1. Starts at index 2
2. Removes 0 elements
3. Inserts {8, 9} at index 2

6.22.3.3. Remove Elements Only

```
tb = {1, 2, 3, 4}
result = Table.splice tb, 2, 3

-- Output
-- tb = {1}
-- result = {2, 3, 4}
```

1. Starts at index 2
2. Removes 3 elements {2, 3, 4}
3. Inserts no new elements

6.23. invert

Tags: **Table only**

invert is used to create a new table with keys and values swapped.

6.23.1. Arguments

Argument	Description	Type	Default
table	table to invert	table	-

6.23.2. Returns

Returns	Description	Type
table	table with key and value inverted	table

6.23.3. Usage

```
tb = { "foo": "bar", "baz": 1 }
result = Table.invert tb

-- Output
--result = {
--  1: "baz"
--  "bar": "foo"
--}
```

6.24. merge

Tags: **Table only**

merge is used to combine key-value tables. The later tables overwrite keys from earlier ones.

6.24.1. Arguments

Argument	Description	Type	Default
...	any number of tables to merge	table	-

6.24.2. Returns

Returns	Description	Type
table	merged table	table

6.24.3. Usage

```
table1 = { a: 1, b: 2 }
table2 = { b: 3, c: 4 }
table3 = { d: 5 }
result = Table.merge table1, table2, table3

-- Output
--result = {
--  a: 1
--  c: 4
--  b: 3
--  d: 5
--}
```

6.25. equal

Tags: List, Table

equal is used to find out if any two tables are equal or not.

6.25.1. Arguments

Argument	Description	Type	Default
first table	first table to compare	table	-
second table	second table to compare	table	

6.25.2. Returns

Returns	Description	Type
state	true if tables are equal, false otherwise	boolean

6.25.3. Usage

```
tbl1 = { a: 1, b: { x: 10, y: 20 }, c: 3 }
tbl2 = { a: 1, b: { x: 10, y: 20 }, c: 3 }
tbl3 = { a: 1, b: { x: 10, y: 30 }, c: 3 }
Table.equal tbl1, tbl2 -- returns true
Table.equal tbl1, tbl3 -- returns false
```

6.26. view

Tags: List, Table

view is used to convert table to string. The produced string is a valid moonscript table. This is used by Logger module and is mostly useful for debugging purposes.

6.26.1. Arguments

Argument	Description	Type	Default
table	table to view	table	-
name	(Optional) name of the table	string	table_unnamed
indent	(Optional) indent each line with a string	string	""

6.26.2. Returns

Returns	Description	Type
table view	string representation of table	string

6.26.3. Usage

```
tb = {1, 2, 3, 4}
result = Table.view tb
```

```
-- Output
-- table_unnamed = {
--   1: 1
--   2: 2
--   3: 3
--   4: 4
-- }
```

```
tb = {"foo": 1, 2, "bar": 3, "baz": 4, 5}
result = Table.view tb, "name"
```

```
-- Output
-- name = {
--   1: 2
--   2: 5
--   "bar": 3
--   "foo": 1
--   "baz": 4
-- }
```