Module:Arguments

From Blue Gold Program Wiki
The printable version is no longer supported and may have rendering errors. Please update your browser bookmarks and please use the default browser print function instead.

40px **This Lua module is used on approximately 24,100,000 pages, or roughly 1352413% of all pages.** To avoid major disruption and server load, any changes should be tested in the module's /sandbox or /testcases subpages, or in your own module sandbox. The tested changes can be added to this page in a single edit. Consider discussing changes on the talk page before implementing them.

40x40px	**This module is used in system messages.** Changes to it can cause immediate changes to the Wikipedia user interface. To avoid large-scale disruption, any changes should first be tested in this module's /sandbox or /testcases subpage, or in your own user space. The tested changes can then be added in one single edit to this module. Please discuss any changes on the talk page before implementing them.
40x40px	This module is subject to page protection. It is a highly visible module in use by a very large number of pages, or is substituted very frequently. Because vandalism or mistakes would affect many pages, and even trivial editing might cause substantial load on the servers, it is protected from editing.

This module provides easy processing of arguments passed from #invoke. It is a meta-module, meant for use by other modules, and should not be called from #invoke directly. Its features include:

- Easy trimming of arguments and removal of blank arguments.
- Arguments can be passed by both the current frame and by the parent frame at the same time. (More details below.)
- Arguments can be passed in directly from another Lua module or from the debug console.
- Arguments are fetched as needed, which can help avoid (some) problems with `<ref>...</ref>` tags.
- Most features can be customized.

□

Contents

Basic use

First, you need to load the module. It contains one function, named `getArgs`.

<source lang="lua"> local getArgs = require('Module:Arguments').getArgs </source>

In the most basic scenario, you can use getArgs inside your main function. The variable `args` is a table containing the arguments from #invoke. (See below for details.)

<source lang="lua"> local getArgs = require('Module:Arguments').getArgs local p = {}

function p.main(frame) local args = getArgs(frame) -- Main module code goes here. end

return p </source>

However, the recommended practice is to use a function just for processing arguments from #invoke. This means that if someone calls your module from another Lua module you don't have to have a frame object available, which improves performance.

<source lang="lua"> local getArgs = require('Module:Arguments').getArgs local p = {}

function p.main(frame) local args = getArgs(frame) return p._main(args) end

function p._main(args) -- Main module code goes here. end

return p </source>

If you want multiple functions to use the arguments, and you also want them to be accessible from #invoke, you can use a wrapper function.

<source lang="lua"> local getArgs = require('Module:Arguments').getArgs

local p = {}

local function makeInvokeFunc(funcName) return function (frame) local args = getArgs(frame) return p[funcName](args) end end

p.func1 = makeInvokeFunc('_func1')

function p._func1(args) -- Code for the first function goes here. end

p.func2 = makeInvokeFunc('_func2')

function p._func2(args) -- Code for the second function goes here. end

return p </source>

Options

The following options are available. They are explained in the sections below.

<source lang="lua"> local args = getArgs(frame, { trim = false, removeBlanks = false, valueFunc = function (key, value) -- Code for processing one argument end, frameOnly = true, parentOnly = true, parentFirst = true, wrappers = { 'Template:A wrapper template', 'Template:Another wrapper template' }, readOnly = true, noOverwrite = true }) </source>

Trimming and removing blanks

Blank arguments often trip up coders new to converting MediaWiki templates to Lua. In template syntax, blank strings and strings consisting only of whitespace are considered false. However, in Lua, blank strings and strings consisting of whitespace are considered true. This means that if you don't pay attention to such arguments when you write your Lua modules, you might treat something as true that should actually be treated as false. To avoid this, by default this module removes all blank arguments.

Similarly, whitespace can cause problems when dealing with positional arguments. Although whitespace is trimmed for named arguments coming from #invoke, it is preserved for positional arguments. Most of the time this additional whitespace is not desired, so this module trims it off by default.

However, sometimes you want to use blank arguments as input, and sometimes you want to keep additional whitespace. This can be necessary to convert some templates exactly as they were written. If you want to do this, you can set the `trim` and `removeBlanks` arguments to `false`.

<source lang="lua"> local args = getArgs(frame, { trim = false, removeBlanks = false }) </source>

Custom formatting of arguments

Sometimes you want to remove some blank arguments but not others, or perhaps you might want to put all of the positional arguments in lower case. To do things like this you can use the `valueFunc` option. The input to this option must be a function that takes two parameters, `key` and `value`, and returns a single value. This value is what you will get when you access the field `key` in the `args` table.

Example 1: this function preserves whitespace for the first positional argument, but trims all other arguments and removes all other blank arguments. <source lang="lua"> local args = getArgs(frame, { valueFunc = function (key, value) if key == 1 then return value elseif value then value = mw.text.trim(value) if value ~= *then* return value end end return nil end }) </source>

Example 2: this function removes blank arguments and converts all arguments to lower case, but doesn't trim whitespace from positional parameters. <source lang="lua"> local args = getArgs(frame, { valueFunc = function (key, value) if not value then return nil end value = mw.ustring.lower(value) if mw.ustring.find(value, '%S') then return value end return nil end }) </source>

Note: the above functions will fail if passed input that is not of type `string` or `nil`. This might be the case if you use the `getArgs` function in the main function of your module, and that function is called by another Lua module. In this case, you will need to check the type of your input. This is not a problem if you are using a function specially for arguments from #invoke (i.e. you have `p.main` and `p._main` functions, or something similar).

Examples 1 and 2 with type checking

Example 1: <source lang="lua"> local args = getArgs(frame, { valueFunc = function (key, value) if key == 1 then return value elseif type(value) == 'string' then value = mw.text.trim(value) if value ~= *then* return value else return nil end else return value end end }) </source>
Example 2: <source lang="lua"> local args = getArgs(frame, { valueFunc = function (key, value) if type(value) == 'string' then value = mw.ustring.lower(value) if mw.ustring.find(value, '%S') then return value else return nil end else return value end end }) </source>

Also, please note that the `valueFunc` function is called more or less every time an argument is requested from the `args` table, so if you care about performance you should make sure you aren't doing anything inefficient with your code.

Frames and parent frames

Arguments in the `args` table can be passed from the current frame or from its parent frame at the same time. To understand what this means, it is easiest to give an example. Let's say that we have a module called `Module:ExampleArgs`. This module prints the first two positional arguments that it is passed.

Module:ExampleArgs code

<source lang="lua"> local getArgs = require('Module:Arguments').getArgs local p = {}
function p.main(frame) local args = getArgs(frame) return p._main(args) end
function p._main(args) local first = args[1] or local second = args[2] or return first .. ' ' .. second
end
return p </source>

`Module:ExampleArgs` is then called by `Template:ExampleArgs`, which contains the code `{{#invoke:ExampleArgs|main|firstInvokeArg}}`. This produces the result "firstInvokeArg".

Now if we were to call `Template:ExampleArgs`, the following would happen:

Code	Result
`{{ExampleArgs}}`	firstInvokeArg
`{{ExampleArgs|firstTemplateArg}}`	firstInvokeArg
`{{ExampleArgs|firstTemplateArg|secondTemplateArg}}`	firstInvokeArg secondTemplateArg

There are three options you can set to change this behaviour: `frameOnly`, `parentOnly` and `parentFirst`. If you set `frameOnly` then only arguments passed from the current frame will be accepted; if you set `parentOnly` then only arguments passed from the parent frame will be accepted; and if you set `parentFirst` then arguments will be passed from both the current and parent frames, but the parent frame will have priority over the current frame. Here are the results in terms of `Template:ExampleArgs`:

frameOnly

Code	Result
`{{ExampleArgs}}`	firstInvokeArg
`{{ExampleArgs|firstTemplateArg}}`	firstInvokeArg
`{{ExampleArgs|firstTemplateArg|secondTemplateArg}}`	firstInvokeArg

parentOnly

Code	Result
`{{ExampleArgs}}`	
`{{ExampleArgs|firstTemplateArg}}`	firstTemplateArg
`{{ExampleArgs|firstTemplateArg|secondTemplateArg}}`	firstTemplateArg secondTemplateArg

parentFirst

Code	Result
`{{ExampleArgs}}`	firstInvokeArg
`{{ExampleArgs|firstTemplateArg}}`	firstTemplateArg
`{{ExampleArgs|firstTemplateArg|secondTemplateArg}}`	firstTemplateArg secondTemplateArg

Notes:

1. If you set both the `frameOnly` and `parentOnly` options, the module won't fetch any

arguments at all from #invoke. This is probably not what you want.
2. In some situations a parent frame may not be available, e.g. if getArgs is passed the parent frame rather than the current frame. In this case, only the frame arguments will be used (unless parentOnly is set, in which case no arguments will be used) and the `parentFirst` and `frameOnly` options will have no effect.

Wrappers

The *wrappers* option is used to specify a limited number of templates as *wrapper templates*, that is, templates whose only purpose is to call a module. If the module detects that it is being called from a wrapper template, it will only check for arguments in the parent frame; otherwise it will only check for arguments in the frame passed to getArgs. This allows modules to be called by either #invoke or through a wrapper template without the loss of performance associated with having to check both the frame and the parent frame for each argument lookup.

For example, the only content of Template:Side box (excluding content in `<noinclude>...</noinclude>` tags) is `{{#invoke:Side box|main}}`. There is no point in checking the arguments passed directly to the #invoke statement for this template, as no arguments will ever be specified there. We can avoid checking arguments passed to #invoke by using the *parentOnly* option, but if we do this then #invoke will not work from other pages either. If this were the case, the `|text=Some text` in the code `{{#invoke:Side box|main|text=Some text}}` would be ignored completely, no matter what page it was used from. By using the `wrappers` option to specify 'Template:Side box' as a wrapper, we can make `{{#invoke:Side box|main|text=Some text}}` work from most pages, while still not requiring that the module check for arguments on the Template:Side box page itself.

Wrappers can be specified either as a string, or as an array of strings.

<source lang="lua"> local args = getArgs(frame, { wrappers = 'Template:Wrapper template' }) </source>

<source lang="lua"> local args = getArgs(frame, { wrappers = { 'Template:Wrapper 1', 'Template:Wrapper 2', -- Any number of wrapper templates can be added here. } }) </source>

Notes:

1. The module will automatically detect if it is being called from a wrapper template's /sandbox subpage, so there is no need to specify sandbox pages explicitly.
2. The *wrappers* option effectively changes the default of the *frameOnly* and *parentOnly* options. If, for example, *parentOnly* were explicitly set to false with *wrappers* set, calls via wrapper templates would result in both frame and parent arguments being loaded, though calls not via wrapper templates would result in only frame arguments being loaded.
3. If the *wrappers* option is set and no parent frame is available, the module will always get the arguments from the frame passed to `getArgs`.

Writing to the args table

Sometimes it can be useful to write new values to the args table. This is possible with the default settings of this module. (However, bear in mind that it is usually better coding style to create a new table with your new values and copy arguments from the args table as needed.)

<source lang="lua"> args.foo = 'some value' </source>

It is possible to alter this behaviour with the `readOnly` and `noOverwrite` options. If `readOnly` is set then it is not possible to write any values to the args table at all. If `noOverwrite` is set, then it is possible to add new values to the table, but it is not possible to add a value if it would overwrite any arguments that are passed from #invoke.

Ref tags

This module uses metatables to fetch arguments from #invoke. This allows access to both the frame arguments and the parent frame arguments without using the `pairs()` function. This can help if your module might be passed `<ref>...</ref>` tags as input.

As soon as `<ref>...</ref>` tags are accessed from Lua, they are processed by the MediaWiki software and the reference will appear in the reference list at the bottom of the article. If your module proceeds to omit the reference tag from the output, you will end up with a phantom reference – a reference that appears in the reference list but without any number linking to it. This has been a problem with modules that use `pairs()` to detect whether to use the arguments from the frame or the parent frame, as those modules automatically process every available argument.

This module solves this problem by allowing access to both frame and parent frame arguments, while still only fetching those arguments when it is necessary. The problem will still occur if you use `pairs(args)` elsewhere in your module, however.

Known limitations

The use of metatables also has its downsides. Most of the normal Lua table tools won't work properly on the args table, including the # operator, the `next()` function, and the functions in the table library. If using these is important for your module, you should use your own argument processing function instead of this module.

```
-- This module provides easy processing of arguments passed to Scribunto from
-- #invoke. It is intended for use by other Lua modules, and should not be
-- called from #invoke directly.

local libraryUtil = require('libraryUtil')
local checkType = libraryUtil.checkType

local arguments = {}

-- Generate four different tidyVal functions, so that we don't have to check
the
-- options every time we call it.

local function tidyValDefault(key, val)
        if type(val) == 'string' then
                val = val:match('^%s*(.-)%s*$')
                if val == '' then
                        return nil
                else
                        return val
```

```
                end
        else
                return val
        end
end

local function tidyValTrimOnly(key, val)
        if type(val) == 'string' then
                return val:match('^%s*(.-)%s*$')
        else
                return val
        end
end

local function tidyValRemoveBlanksOnly(key, val)
        if type(val) == 'string' then
                if val:find('%S') then
                        return val
                else
                        return nil
                end
        else
                return val
        end
end

local function tidyValNoChange(key, val)
        return val
end

local function matchesTitle(given, title)
        local tp = type( given )
        return (tp == 'string' or tp == 'number') and mw.title.new( given
).prefixedText == title
end

local translate_mt = { __index = function(t, k) return k end }

function arguments.getArgs(frame, options)
        checkType('getArgs', 1, frame, 'table', true)
        checkType('getArgs', 2, options, 'table', true)
        frame = frame or {}
        options = options or {}

        --[[
        -- Set up argument translation.
        --]]
        options.translate = options.translate or {}
        if getmetatable(options.translate) == nil then
                setmetatable(options.translate, translate_mt)
        end
```

```
        if options.backtranslate == nil then
                options.backtranslate = {}
                for k,v in pairs(options.translate) do
                        options.backtranslate[v] = k
                end
        end
        if options.backtranslate and getmetatable(options.backtranslate) ==
nil then
                setmetatable(options.backtranslate, {
                        __index = function(t, k)
                                if options.translate[k] ~= k then
                                        return nil
                                else
                                        return k
                                end
                        end
                })
        end

        --[[
        -- Get the argument tables. If we were passed a valid frame object,
get the
        -- frame arguments (fargs) and the parent frame arguments (pargs),
depending
        -- on the options set and on the parent frame's availability. If we
weren't
        -- passed a valid frame object, we are being called from another Lua
module
        -- or from the debug console, so assume that we were passed a table
of args
        -- directly, and assign it to a new variable (luaArgs).
        --]]
        local fargs, pargs, luaArgs
        if type(frame.args) == 'table' and type(frame.getParent) ==
'function' then
                if options.wrappers then
                        --[[
                        -- The wrappers option makes Module:Arguments look up
arguments in
                        -- either the frame argument table or the parent
argument table, but
                        -- not both. This means that users can use either the
#invoke syntax
                        -- or a wrapper template without the loss of
performance associated
                        -- with looking arguments up in both the frame and
the parent frame.
                        -- Module:Arguments will look up arguments in the
parent frame
                        -- if it finds the parent frame's title in
options.wrapper;
```

```
                        -- otherwise it will look up arguments in the frame
object passed
                        -- to getArgs.
                        --]]
                        local parent = frame:getParent()
                        if not parent then
                                fargs = frame.args
                        else
                                local title =
parent:getTitle():gsub('/sandbox$', '')
                                local found = false
                                if matchesTitle(options.wrappers, title) then
                                        found = true
                                elseif type(options.wrappers) == 'table' then
                                        for _,v in pairs(options.wrappers) do
                                                if matchesTitle(v, title)
then
                                                        found = true
                                                        break
                                                end
                                        end
                                end

                                -- We test for false specifically here so
that nil (the default) acts like true.
                                if found or options.frameOnly == false then
                                        pargs = parent.args
                                end
                                if not found or options.parentOnly == false
then
                                        fargs = frame.args
                                end
                        end
                else
                        -- options.wrapper isn't set, so check the other
options.
                        if not options.parentOnly then
                                fargs = frame.args
                        end
                        if not options.frameOnly then
                                local parent = frame:getParent()
                                pargs = parent and parent.args or nil
                        end
                end
                if options.parentFirst then
                        fargs, pargs = pargs, fargs
                end
        else
                luaArgs = frame
        end
```

```
        -- Set the order of precedence of the argument tables. If the
variables are
        -- nil, nothing will be added to the table, which is how we avoid
clashes
        -- between the frame/parent args and the Lua args.
        local argTables = {fargs}
        argTables[#argTables + 1] = pargs
        argTables[#argTables + 1] = luaArgs

        --[[
        -- Generate the tidyVal function. If it has been specified by the
user, we
        -- use that; if not, we choose one of four functions depending on the
        -- options chosen. This is so that we don't have to call the options
table
        -- every time the function is called.
        --]]
        local tidyVal = options.valueFunc
        if tidyVal then
                if type(tidyVal) ~= 'function' then
                        error(
                                "bad value assigned to option 'valueFunc'"
                                        .. '(function expected, got '
                                        .. type(tidyVal)
                                        .. ')',
                                2
                        )
                end
        elseif options.trim ~= false then
                if options.removeBlanks ~= false then
                        tidyVal = tidyValDefault
                else
                        tidyVal = tidyValTrimOnly
                end
        else
                if options.removeBlanks ~= false then
                        tidyVal = tidyValRemoveBlanksOnly
                else
                        tidyVal = tidyValNoChange
                end
        end

        --[[
        -- Set up the args, metaArgs and nilArgs tables. args will be the one
        -- accessed from functions, and metaArgs will hold the actual
arguments. Nil
        -- arguments are memoized in nilArgs, and the metatable connects all
of them
        -- together.
        --]]
        local args, metaArgs, nilArgs, metatable = {}, {}, {}, {}
```

```
	setmetatable(args, metatable)

	local function mergeArgs(tables)
		--[[
		-- Accepts multiple tables as input and merges their keys and
values
		-- into one table. If a value is already present it is not
overwritten;
		-- tables listed earlier have precedence. We are also
memoizing nil
		-- values, which can be overwritten if they are 's' (soft).
		--]]
		for _, t in ipairs(tables) do
			for key, val in pairs(t) do
				if metaArgs[key] == nil and nilArgs[key] ~=
'h' then
					local tidiedVal = tidyVal(key, val)
					if tidiedVal == nil then
						nilArgs[key] = 's'
					else
						metaArgs[key] = tidiedVal
					end
				end
			end
		end
	end

	--[[
	-- Define metatable behaviour. Arguments are memoized in the metaArgs
table,
	-- and are only fetched from the argument tables once. Fetching
arguments
	-- from the argument tables is the most resource-intensive step in
this
	-- module, so we try and avoid it where possible. For this reason,
nil
	-- arguments are also memoized, in the nilArgs table. Also, we keep a
record
	-- in the metatable of when pairs and ipairs have been called, so we
do not
	-- run pairs and ipairs on the argument tables more than once. We
also do
	-- not run ipairs on fargs and pargs if pairs has already been run,
as all
	-- the arguments will already have been copied over.
	--]]

	metatable.__index = function (t, key)
		--[[
		-- Fetches an argument when the args table is indexed. First
we check
```

```
                -- to see if the value is memoized, and if not we try and
fetch it from
                -- the argument tables. When we check memoization, we need to
check
                -- metaArgs before nilArgs, as both can be non-nil at the
same time.
                -- If the argument is not present in metaArgs, we also check
whether
                -- pairs has been run yet. If pairs has already been run, we
return nil.
                -- This is because all the arguments will have already been
copied into
                -- metaArgs by the mergeArgs function, meaning that any other
arguments
                -- must be nil.
                --]]
                if type(key) == 'string' then
                        key = options.translate[key]
                end
                local val = metaArgs[key]
                if val ~= nil then
                        return val
                elseif metatable.donePairs or nilArgs[key] then
                        return nil
                end
                for _, argTable in ipairs(argTables) do
                        local argTableVal = tidyVal(key, argTable[key])
                        if argTableVal ~= nil then
                                metaArgs[key] = argTableVal
                                return argTableVal
                        end
                end
                nilArgs[key] = 'h'
                return nil
        end

        metatable.__newindex = function (t, key, val)
                -- This function is called when a module tries to add a new
value to the
                -- args table, or tries to change an existing value.
                if type(key) == 'string' then
                        key = options.translate[key]
                end
                if options.readOnly then
                        error(
                                'could not write to argument table key "'
                                        .. tostring(key)
                                        .. '"; the table is read-only',
                                2
                        )
                elseif options.noOverwrite and args[key] ~= nil then
```

```
                        error(
                                'could not write to argument table key "'
                                        .. tostring(key)
                                        .. '"; overwriting existing arguments
is not permitted',
                                2
                        )
                elseif val == nil then
                        --[[
                        -- If the argument is to be overwritten with nil, we
need to erase
                        -- the value in metaArgs, so that __index, __pairs
and __ipairs do
                        -- not use a previous existing value, if present; and
we also need
                        -- to memoize the nil in nilArgs, so that the value
isn't looked
                        -- up in the argument tables if it is accessed again.
                        --]]
                        metaArgs[key] = nil
                        nilArgs[key] = 'h'
                else
                        metaArgs[key] = val
                end
        end

        local function translatenext(invariant)
                local k, v = next(invariant.t, invariant.k)
                invariant.k = k
                if k == nil then
                        return nil
                elseif type(k) ~= 'string' or not options.backtranslate then
                        return k, v
                else
                        local backtranslate = options.backtranslate[k]
                        if backtranslate == nil then
                                -- Skip this one. This is a tail call, so
this won't cause stack overflow
                                return translatenext(invariant)
                        else
                                return backtranslate, v
                        end
                end
        end

        metatable.__pairs = function ()
                -- Called when pairs is run on the args table.
                if not metatable.donePairs then
                        mergeArgs(argTables)
                        metatable.donePairs = true
                end
```

```
                return translatenext, { t = metaArgs }
        end

        local function inext(t, i)
                -- This uses our __index metamethod
                local v = t[i + 1]
                if v ~= nil then
                        return i + 1, v
                end
        end

        metatable.__ipairs = function (t)
                -- Called when ipairs is run on the args table.
                return inext, t, 0
        end

        return args
end

return arguments
```

Retrieved from "https://www.bluegoldwiki.com/index.php?title=Module:Arguments&oldid=2265"

Namespaces

- Module
- Discussion

Variants

Categories:

- Pages with script errors
- Pages with broken file links
- Pages used in system messages needing protection
- Modules subject to page protection
- Lua metamodules

This page was last edited on 26 April 2020, at 05:11.

Blue Gold Program Wiki

The wiki version of the Lessons Learnt Report of the Blue Gold program, documents the experiences of a technical assistance (TA) team working in a development project implemented by the Bangladesh Water Development Board (BWDB) and the Department of Agricultural Extension (DAE) over an eight+ year period from March 2013 to December 2021. The wiki lessons learnt report (LLR) is intended to complement the BWDB and DAE project completion reports (PCRs), with the aim of recording lessons learnt for use in the design and implementation of future interventions in the coastal zone.

- About Blue Gold Program Wiki
- Disclaimers

Developed and maintained by Big Blue Communications for Blue Gold Program

- Powered by MediaWiki

Blue Gold Program Wiki