



Inhaltsverzeichnis

1. Modul:Hatnote list	2
2. Modul:Arguments	7
3. Modul:Hatnote	21

Modul:Hatnote list

Vorlage:Used in system

Inhaltsverzeichnis	
1 Usage from wikitext	2
2 Usage from Lua	2
2.1 andList	2
2.2 orList	2
2.3 forSee	2

Usage from wikitext

This module is not designed be used directly from wikitext even though `forSee` does take an initial frame argument and could potentially be used from wikitext, e.g.:

- `{{hatnote|PREFIX {{#invoke:Hatnote list|forSee|{{tl|For}}|Module:For|{{tl|About}}|Module:About}} POSTFIX}}` → [Vorlage:Hatnote](#)

Usage from Lua

To call the module, use

```
local mHatList = require('Module:Hatnote list')
```

or similar, then access its methods through the `mHatList` variable (or whatever was used).

andList

`andList` takes a list in table form, and returns a string with the list separated with "and" and commas as appropriate.

orList

`orList` takes a list in table form, and returns a string with the list separated with "or" and commas as appropriate.

forSee

`_forSee` takes three arguments: a table of trimmed arguments with blanks removed, a "from" number with the index to start at, and an options table, and returns a string with a number of "For X, see [[Y]]" sentences. The links are formatted using the methods from [Module:Hatnote](#).



As a convenience, the `forSee` method (without the leading underscore) takes the same arguments except with a frame instead of an args table, using `getArgs()` from `Module:Arguments` to preprocess the arguments.

```
-----
--                                     Module:Hatnote list                                     --
--                                     -----                                     --
-- This module produces and formats lists for use in hatnotes. In particular, --
-- it implements the for-see list, i.e. lists of "For X, see Y" statements, --
-- as used in {{about}}, {{redirect}}, and their variants. Also introduced --
-- are andList & orList helpers for formatting lists with those conjunctions. --
-----

local mArguments --initialize lazily
local mFormatLink = require('Module:Format link')
local mHatnote = require('Module:Hatnote')
local libraryUtil = require('libraryUtil')
local checkType = libraryUtil.checkType
local p = {}

-----

-- List stringification helper functions
--
-- These functions are used for stringifying lists, usually page lists inside
-- the "Y" portion of "For X, see Y" for-see items.
-----

--default options table used across the list stringification functions
local stringifyListDefaultOptions = {
    conjunction = "and",
    separator = ", ",
    altSeparator = ";",
    space = " ",
    formatted = false
}

--Searches display text only
local function searchDisp(haystack, needle)
    return string.find(
        string.sub(haystack, (string.find(haystack, '|') or 0) + 1), needle
    )
end

-- Stringifies a list generically; probably shouldn't be used directly
local function stringifyList(list, options)
    -- Type-checks, defaults, and a shortcut
    checkType("stringifyList", 1, list, "table")
    if #list == 0 then return nil end
    checkType("stringifyList", 2, options, "table", true)
    options = options or {}
    for k, v in pairs(stringifyListDefaultOptions) do
        if options[k] == nil then options[k] = v end
    end
    local s = options.space
    -- Format the list if requested
    if options.formatted then
        list = mFormatLink.formatPages(
            {categorizeMissing = mHatnote.missingTargetCat}, list
        )
    end
    -- Set the separator; if any item contains it, use the alternate separate
end
```

```
    local separator = options.separator
    for k, v in pairs(list) do
        if searchDisp(v, separator) then
            separator = options.altSeparator
            break
        end
    end
end
-- Set the conjunction, apply Oxford comma, and force a comma if #1 has
local conjunction = s .. options.conjunction .. s
if #list == 2 and searchDisp(list[1], "$") or #list > 2 then
    conjunction = separator .. conjunction
end
-- Return the formatted string
return mw.text.listToText(list, separator .. s, conjunction)
end

--DRY function
function p.conjList (conj, list, fmt)
    return stringifyList(list, {conjunction = conj, formatted = fmt})
end

-- Stringifies lists with "and" or "or"
function p.andList (...) return p.conjList("and", ...) end
function p.orList (...) return p.conjList("or", ...) end

-----
-- For see
--
-- Makes a "For X, see [[Y]]." list from raw parameters. Intended for the
-- {{about}} and {{redirect}} templates and their variants.
-----

--default options table used across the forSee family of functions
local forSeeDefaultOptions = {
    andKeyword = 'and',
    title = mw.title.getCurrentTitle().text,
    otherText = 'other uses',
    forSeeForm = 'For %s, see %s.',
}

--Collapses duplicate punctuation
local function punctuationCollapse (text)
    local replacements = {
        ["%.%.$"] = ".",
        ["%?%.$"] = "?",
        ["%!%.$"] = "!",
        ["%.%]%"%.$"] = ".]]",
        ["%?%]%"%.$"] = "?]]",
        ["%!%]%"%.$"] = "!]]"
    }
    for k, v in pairs(replacements) do text = string.gsub(text, k, v) end
    return text
end

-- Structures arguments into a table for stringification, & options
function p.forSeeArgsToTable (args, from, options)
    -- Type-checks and defaults
    checkType("forSeeArgsToTable", 1, args, 'table')
    checkType("forSeeArgsToTable", 2, from, 'number', true)
    from = from or 1
    checkType("forSeeArgsToTable", 3, options, 'table', true)
    options = options or {}
    for k, v in pairs(forSeeDefaultOptions) do
        if options[k] == nil then options[k] = v end
    end
end
```

```
end
-- maxArg's gotten manually because getArgs() and table.maxn aren't friend
local maxArg = 0
for k, v in pairs(args) do
    if type(k) == 'number' and k > maxArg then maxArg = k end
end
-- Structure the data out from the parameter list:
-- * forTable is the wrapper table, with forRow rows
-- * Rows are tables of a "use" string & a "pages" table of pagename strings
-- * Blanks are left empty for defaulting elsewhere, but can terminate list
local forTable = {}
local i = from
local terminated = false
-- If there is extra text, and no arguments are given, give nil value
-- to not produce default of "For other uses, see foo (disambiguation)"
if options.extratext and i > maxArg then return nil end
-- Loop to generate rows
repeat
    -- New empty row
    local forRow = {}
    -- On blank use, assume list's ended & break at end of this loop
    forRow.use = args[i]
    if not args[i] then terminated = true end
    -- New empty list of pages
    forRow.pages = {}
    -- Insert first pages item if present
    table.insert(forRow.pages, args[i + 1])
    -- If the param after next is "and", do inner loop to collect pages
    -- until the "and"'s stop. Blanks are ignored: "1|and||and|3" → {1,3}
    while args[i + 2] == options.andKeyword do
        if args[i + 3] then
            table.insert(forRow.pages, args[i + 3])
        end
        -- Increment to next "and"
        i = i + 2
    end
    -- Increment to next use
    i = i + 2
    -- Append the row
    table.insert(forTable, forRow)
until terminated or i > maxArg

return forTable
end

-- Stringifies a table as formatted by forSeeArgsToTable
function p.forSeeTableToString (forSeeTable, options)
    -- Type-checks and defaults
    checkType("forSeeTableToString", 1, forSeeTable, "table", true)
    checkType("forSeeTableToString", 2, options, "table", true)
    options = options or {}
    for k, v in pairs(forSeeDefaultOptions) do
        if options[k] == nil then options[k] = v end
    end
    -- Stringify each for-see item into a list
    local strList = {}
    if forSeeTable then
        for k, v in pairs(forSeeTable) do
            local useStr = v.use or options.otherText
            local pagesStr =
                p.andList(v.pages, true) or
                mFormatLink._formatLink{
                    categorizeMissing = mHatnote.missingTarget,
                    link = mHatnote.disambiguate(options.title, v)
                }
            strList[k] = useStr .. pagesStr
        end
    end
end
```



```
        }
        local forSeeStr = string.format(options.forSeeForm, useSt
forSeeStr = punctuationCollapse(forSeeStr)
table.insert(strList, forSeeStr)
    end
end
if options.extratext then table.insert(strList, punctuationCollapse(optic
-- Return the concatenated list
return table.concat(strList, ' ')
end

-- Produces a "For X, see [[Y]]" string from arguments. Expects index gaps
-- but not blank/whitespace values. Ignores named args and args < "from".
function p._forSee (args, from, options)
    local forSeeTable = p.forSeeArgsToTable(args, from, options)
    return p.forSeeTableToString(forSeeTable, options)
end

-- As _forSee, but uses the frame.
function p.forSee (frame, from, options)
    mArguments = require('Module:Arguments')
    return p._forSee(mArguments.getArgs(frame), from, options)
end

return p
```

Modul:Arguments

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.
- Most features can be customized.

Inhaltsverzeichnis

1 Basic use	7
1.1 Recommended practice	8
1.2 Multiple functions	8
1.3 Options	9
1.4 Trimming and removing blanks	9
1.5 Custom formatting of arguments	9
1.6 Frames and parent frames	11
1.7 Wrappers	13
1.8 Writing to the args table	14
1.9 Ref tags	14
1.10 Known limitations	14

Basic use

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

```
local getArgs = require('Module:Arguments').getArgs
```

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.)

```
local getArgs = require('Module:Arguments').getArgs
local p = {}

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

return p
```

Recommended practice

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.

```
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
```

The way this is called from a template is `{{#invoke:Example|main}}` (optionally with some parameters like `{{#invoke:Example|main|arg1=value1|arg2=value2}}`), and the way this is called from a module is `require('Module:Example')._main({arg1 = 'value1', arg2 = value2, 'spaced arg3' = 'value3'})`. What this second one does is construct a table with the arguments in it, then gives that table to the `p._main(args)` function, which uses it natively.

Multiple functions

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.

```
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
```

Options

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

```
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
})
```

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`.

```
local args = getArgs(frame, {
  trim = false,
  removeBlanks = false
})
```

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.

```
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
})
```

Example 2: this function removes blank arguments and converts all arguments to lower case, but doesn't trim whitespace from positional parameters.

```
local args = getArgs(frame, {
valueFunc = function (key, value)
if not value then
return nil
end
value = mw.usttring.lower(value)
if mw.usttring.find(value, '%S') then
return value
end
return nil
end
})
```

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).

Vorlage:Cot Example 1:

```
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
end
end
})
```



```
end
else
return value
end
end
})
```

Example 2:

```
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
})
```

Vorlage:Cob

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.

Vorlage:Cot

```
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
```

Vorlage:Cob

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
<code>{{ExampleArgs}}</code>	firstInvokeArg
<code>{{ExampleArgs firstTemplateArg}}</code>	firstInvokeArg
<code>{{ExampleArgs firstTemplateArg secondTemplateArg}}</code>	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
<code>{{ExampleArgs}}</code>	firstInvokeArg
<code>{{ExampleArgs firstTemplateArg}}</code>	firstInvokeArg
<code>{{ExampleArgs firstTemplateArg secondTemplateArg}}</code>	firstInvokeArg

parentOnly

Code	Result
<code>{{ExampleArgs}}</code>	
<code>{{ExampleArgs firstTemplateArg}}</code>	firstTemplateArg
<code>{{ExampleArgs firstTemplateArg secondTemplateArg}}</code>	firstTemplateArg secondTemplateArg

parentFirst

Code	Result
<code>{{ExampleArgs}}</code>	firstInvokeArg
<code>{{ExampleArgs firstTemplateArg}}</code>	firstTemplateArg
<code>{{ExampleArgs firstTemplateArg secondTemplateArg}}</code>	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 [Vorlage:Tag](#) 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 [Vorlage:Para](#) 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.

```
local args = getArgs(frame, {
wrappers = 'Template:Wrapper template'
})
```

```
local args = getArgs(frame, {
wrappers = {
'Template:Wrapper 1',
'Template:Wrapper 2',
-- Any number of wrapper templates can be added here.
}
})
```

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 0 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.)

```
args.foo = 'some value'
```

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 `Vorlage:Tag` tags as input.

As soon as `Vorlage:Tag` 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 ).prefi
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 {}
end
```



```
--[[
-- 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 were
-- 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 arguments
-- 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 #if argument
        -- or a wrapper template without the loss of performance.
        -- with looking arguments up in both the frame and the parent.
        -- Module:Arguments will look up arguments in the parent table
        -- if it finds the parent frame's title in options.wrappers,
        -- otherwise it will look up arguments in the frame object
        -- 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
        end
    end
end
```

```
-- We test for false specifically here so that nil
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

end

--[[
```

```
-- Set up the args, metaArgs and nilArgs tables. args will be the one
-- accessed from functions, and metaArgs will hold the actual arguments.
-- arguments are memoized in nilArgs, and the metatable connects all of
-- 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 the
  -- 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

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 over
  -- metaArgs by the mergeArgs function, meaning that any other argument
  -- 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])
  end
end
```

```
        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
    -- 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
            2
        )
    elseif val == nil then
        --[[
        -- If the argument is to be overwritten with nil, we need
        -- the value in metaArgs, so that __index, __pairs and __
        -- not use a previous existing value, if present; and we
        -- to memoize the nil in nilArgs, so that the value isn't
        -- 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 wo
            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
```

Modul:Hatnote

Vorlage:Lua

This is a meta-module that provides various functions for making [hatnotes](#). It implements the [Vorlage:TI](#) template, for use in hatnotes at the top of pages, and the [Vorlage:TI](#) template, which is used to format a wikilink for use in hatnotes. It also contains a number of helper functions for use in other Lua hatnote modules.

Inhaltsverzeichnis

1 Use from wikitext	21
2 Use from other Lua modules	21
2.1 Hatnote	21
2.2 Find namespace id	22
2.3 Make wikitext error	22
3 Examples	22

Use from wikitext

The functions in this module cannot be used directly from `#invoke`, and must be used through templates instead. Please see [Template:Hatnote](#) and [Template:Format link](#) for documentation.

Use from other Lua modules

To load this module from another Lua module, use the following code.

```
local mHatnote = require('Module:Hatnote')
```

You can then use the functions as documented below.

Hatnote

```
mHatnote._hatnote(s, options)
```

Formats the string *s* as a hatnote. This encloses *s* in the tags [Vorlage:Tag](#). Options are provided in the *options* table. Options include:

- *options.extraclasses* - a string of extra classes to provide
- *options.selfref* - if this is not nil or false, adds the class "selfref", used to denote self-references to Wikipedia (see [Template:Selfref](#))

The CSS of the hatnote class is defined in [Module:Hatnote/styles.css](#).

Example 1



```
mHatnote._hatnote('This is a hatnote.')
```

Produces: **Vorlage:Tag**

Displays as: **Vorlage:Hatnote**

Example 2

```
mHatnote._hatnote('This is a hatnote.', {extraclasses = 'boilerplate seealso', se
```

Produces: **Vorlage:Tag**

Displayed as: **Vorlage:Hatnote**

Find namespace id

```
mHatnote.findNamespaceId(link, removeColon)
```

Finds the **namespace id** of the string *link*, which should be a valid page name, with or without the section name. This function will not work if the page name is enclosed with square brackets. When trying to parse the namespace name, colons are removed from the start of the link by default. This is helpful if users have specified colons when they are not strictly necessary. If you do not need to check for initial colons, set *removeColon* to false.

Examples

Vorlage:Code → 0

Vorlage:Code → 14

Vorlage:Code → 14

Vorlage:Code → 0 (the namespace is detected as ":Category", rather than "Category")

Make wikitext error

```
mHatnote.makeWikitextError(msg, helpLink, addTrackingCategory)
```

Formats the string *msg* as a red wikitext error message, with optional link to a help page *helpLink*. Normally this function also adds **Vorlage:Clc**. To suppress categorization, pass `false` as third parameter of the function (`addTrackingCategory`).

Examples:

Vorlage:Code → **Error: an error has occurred.**

Vorlage:Code → **Error: an error has occurred (help).**

Examples

For an example of how this module is used in other Lua modules, see **Module:Main**

```
-----
--                                     Module:Hatnote                                     --
--                                     -----                                     --
-- This module produces hatnote links and links to related articles. It         --
-- implements the {{hatnote}} and {{format link}} meta-templates and includes --
-- helper functions for other Lua hatnote modules.                             --
-----

local libraryUtil = require('libraryUtil')
local checkType = libraryUtil.checkType
local checkTypeForNamedArg = libraryUtil.checkTypeForNamedArg
local mArguments -- lazily initialise [[Module:Arguments]]
local yesno -- lazily initialise [[Module:Yesno]]
local formatLink -- lazily initialise [[Module:Format link]] ._formatLink

local p = {}

-----
-- Helper functions
-----

local curNs = mw.title.getCurrentTitle().namespace
p.missingTargetCat =
    --Default missing target category, exported for use in related modules
    ((curNs == 0) or (curNs == 14)) and
    'Articles with hatnote templates targeting a nonexistent page' or nil

local function getArgs(frame)
    -- Fetches the arguments from the parent frame. Whitespace is trimmed and
    -- blanks are removed.
    mArguments = require('Module:Arguments')
    return mArguments.getArgs(frame, {parentOnly = true})
end

local function removeInitialColon(s)
    -- Removes the initial colon from a string, if present.
    return s:match('^:?(.*)')
end

function p.findNamespaceId(link, removeColon)
    -- Finds the namespace id (namespace number) of a link or a pagename. This
    -- function will not work if the link is enclosed in double brackets. Colons
    -- are trimmed from the start of the link by default. To skip colon
    -- trimming, set the removeColon parameter to false.
    checkType('findNamespaceId', 1, link, 'string')
    checkType('findNamespaceId', 2, removeColon, 'boolean', true)
    if removeColon ~= false then
        link = removeInitialColon(link)
    end
    local namespace = link:match('^(:?)(.*):')
    if namespace then
        local nsTable = mw.site.namespaces[namespace]
        if nsTable then
            return nsTable.id
        end
    end
    return 0
end

function p.makeWikitextError(msg, helpLink, addTrackingCategory, title)
    -- Formats an error message to be returned to wikitext. If
    -- addTrackingCategory is not false after being returned from
    -- [[Module:Yesno]], and if we are not on a talk page, a tracking category
    -- is added.
```

```
    checkType('makeWikitextError', 1, msg, 'string')
    checkType('makeWikitextError', 2, helpLink, 'string', true)
    yesno = require('Module:Yesno')
    title = title or mw.title.getCurrentTitle()
    -- Make the help link text.
    local helpText
    if helpLink then
        helpText = ' ([[ ' .. helpLink .. '|help]])'
    else
        helpText = ''
    end
    -- Make the category text.
    local category
    if not title.isTalkPage -- Don't categorise talk pages
        and title.namespace ~= 2 -- Don't categorise userspace
        and yesno(addTrackingCategory) ~= false -- Allow opting out
    then
        category = 'Hatnote templates with errors'
        category = mw.usttring.format(
            '[[%s:%s]]',
            mw.site.namespaces[14].name,
            category
        )
    else
        category = ''
    end
    return mw.usttring.format(
        '<strong class="error">Error: %s%s.</strong>%s',
        msg,
        helpText,
        category
    )
end

function p.disambiguate(page, disambiguator)
    -- Formats a page title with a disambiguation parenthetical,
    -- i.e. "Example" → "Example (disambiguation)".
    checkType('disambiguate', 1, page, 'string')
    checkType('disambiguate', 2, disambiguator, 'string', true)
    disambiguator = disambiguator or 'disambiguation'
    return mw.usttring.format('%s (%s)', page, disambiguator)
end

-----
-- Hatnote
--
-- Produces standard hatnote text. Implements the {{hatnote}} template.
-----

function p.hatnote(frame)
    local args = getArgs(frame)
    local s = args[1]
    if not s then
        return p.makeWikitextError(
            'no text specified',
            'Template:Hatnote#Errors',
            args.category
        )
    end
    return p._hatnote(s, {
        extraclasses = args.extraclasses,
        selfref = args.selfref
    })
end
```

```
function p._hatnote(s, options)
  checkType('_hatnote', 1, s, 'string')
  checkType('_hatnote', 2, options, 'table', true)
  options = options or {}
  local inline = options.inline
  local hatnote = mw.html.create(inline == 1 and 'span' or 'div')
  local extraclasses
  if type(options.extraclasses) == 'string' then
    extraclasses = options.extraclasses
  end

  hatnote
    :attr('role', 'note')
    :addClass(inline == 1 and 'hatnote-inline' or 'hatnote')
    :addClass('navigation-not-searchable')
    :addClass(extraclasses)
    :addClass(options.selfref and 'selfref')
    :wikitext(s)

  return mw.getCurrentFrame():extensionTag{
    name = 'templatestyles', args = { src = 'Module:Hatnote/styles.cs
  } .. tostring(hatnote)
end

return p
```