



Inhaltsverzeichnis

1. Modul:Yesno/Doku	2
2. Modul:Arguments	5
3. Modul:Yesno	19

Modul:Yesno/Doku

Dies ist die Dokumentationsseite für Modul:Yesno

This module provides a consistent interface for processing boolean or boolean-style string input. While Lua allows the `true` and `false` boolean values, wikicode templates can only express boolean values through strings such as "yes", "no", etc. This module processes these kinds of strings and turns them into boolean input for Lua to process. It also returns `nil` values as `nil`, to allow for distinctions between `nil` and `false`. The module also accepts other Lua structures as input, i.e. booleans, numbers, tables, and functions. If it is passed input that it does not recognise as boolean or `nil`, it is possible to specify a default value to return.

Inhaltsverzeichnis

1 Syntax	2
2 Usage	2
2.1 Undefined input ('foo')	3
2.2 Handling nil results	4

Syntax

```
yesno(value, default)
```

`value` is the value to be tested. Boolean input or boolean-style input (see below) always evaluates to either `true` or `false`, and `nil` always evaluates to `nil`. Other values evaluate to `default`.

Usage

First, load the module. Note that it can only be loaded from other Lua modules, not from normal wiki pages. For normal wiki pages you can use [Vorlage:TI](#) instead.

```
local yesno = require('Module:Yesno')
```

Some input values always return `true`, and some always return `false`. `nil` values always return `nil`.

```
-- These always return true:  
yesno('yes')  
yesno('y')  
yesno('true')  
yesno('t')  
yesno('1')  
yesno(1)  
yesno(true)
```



```
-- These always return false:  
yesno('no')  
yesno('n')  
yesno('false')  
yesno('f')  
yesno('0')  
yesno(0)  
yesno(false)  
  
-- A nil value always returns nil:  
yesno(nil)
```

String values are converted to lower case before they are matched:

```
-- These always return true:  
yesno('Yes')  
yesno('YES')  
yesno('yEs')  
yesno('Y')  
yesno('tRuE')  
  
-- These always return false:  
yesno('No')  
yesno('NO')  
yesno('n0')  
yesno('N')  
yesno('fALsE')
```

Undefined input ('foo')

You can specify a default value if `yesno` receives input other than that listed above. If you don't supply a default, the module will return `nil` for these inputs.

```
-- These return nil:  
yesno('foo')  
yesno({})  
yesno(5)  
yesno(function() return 'This is a function.' end)  
yesno(nil, true)  
yesno(nil, 'bar')  
  
-- These return true:  
yesno('foo', true)  
yesno({}, true)  
yesno(5, true)  
yesno(function() return 'This is a function.' end, true)  
  
-- These return "bar":  
yesno('foo', 'bar')  
yesno({}, 'bar')  
yesno(5, 'bar')  
yesno(function() return 'This is a function.' end, 'bar')
```

Note that the empty string also functions this way:



```
yesno('')          -- Returns nil.  
yesno('', true)   -- Returns true.  
yesno('', 'bar')  -- Returns "bar".
```

Although the empty string usually evaluates to false in wikitext, it evaluates to true in Lua. This module prefers the Lua behaviour over the wikitext behaviour. If treating the empty string as false is important for your module, you will need to convert empty strings to a value that evaluates to false before passing them to this module. In the case of arguments received from wikitext, this can be done by using [Module:Arguments](#).

Handling nil results

By definition

```
yesno(nil)          -- Returns nil.  
yesno('foo')       -- Returns nil.  
yesno(nil, true)   -- Returns nil.  
yesno(nil, false)  -- Returns nil.  
yesno('foo', true) -- Returns true.
```

To get the binary true/false-only values, use code like:

```
myvariable = yesno(value) or false -- When value is nil, result is false.  
myvariable = yesno(value) or true  -- When value is nil, result is true.  
myvariable = yesno('foo') or false -- Unknown string returns nil, result is false.  
myvariable = yesno('foo', true) or false -- Default value (here: true) applies,
```

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	5
1.1 Recommended practice	6
1.2 Multiple functions	6
1.3 Options	7
1.4 Trimming and removing blanks	7
1.5 Custom formatting of arguments	7
1.6 Frames and parent frames	9
1.7 Wrappers	11
1.8 Writing to the args table	12
1.9 Ref tags	12
1.10 Known limitations	12

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 t
-- together.
--]]
local args, metaArgs, nilArgs, metatable = {}, {}, {}, {}
setmetatable(args, metatable)

local function mergeArgs(tables)
  --[[
  -- Accepts multiple tables as input and merges their keys and val
  -- into one table. If a value is already present it is not overw
  -- tables listed earlier have precedence. We are also memoizing r
  -- 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' t
        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 ta
-- 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 rec
-- in the metatable of when pairs and ipairs have been called, so we do n
-- 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 al
-- the arguments will already have been copied over.
--]]

metatable.__index = function (t, key)
  --[[
  -- Fetches an argument when the args table is indexed. First we c
  -- to see if the value is memoized, and if not we try and fetch i
  -- the argument tables. When we check memoization, we need to che
  -- metaArgs before nilArgs, as both can be non-nil at the same ti
  -- If the argument is not present in metaArgs, we also check whet
  -- pairs has been run yet. If pairs has already been run, we retu
  -- This is because all the arguments will have already been copie
  -- metaArgs by the mergeArgs function, meaning that any other arg
  -- 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
    -- 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:Yesno

This module provides a consistent interface for processing boolean or boolean-style string input. While Lua allows the `true` and `false` boolean values, wikicode templates can only express boolean values through strings such as "yes", "no", etc. This module processes these kinds of strings and turns them into boolean input for Lua to process. It also returns `nil` values as `nil`, to allow for distinctions between `nil` and `false`. The module also accepts other Lua structures as input, i.e. booleans, numbers, tables, and functions. If it is passed input that it does not recognise as boolean or `nil`, it is possible to specify a default value to return.

Inhaltsverzeichnis

1 Syntax	19
2 Usage	19
2.1 Undefined input ('foo')	20
2.2 Handling nil results	21

Syntax

```
yesno(value, default)
```

`value` is the value to be tested. Boolean input or boolean-style input (see below) always evaluates to either `true` or `false`, and `nil` always evaluates to `nil`. Other values evaluate to `default`.

Usage

First, load the module. Note that it can only be loaded from other Lua modules, not from normal wiki pages. For normal wiki pages you can use [Vorlage:TI](#) instead.

```
local yesno = require('Module:Yesno')
```

Some input values always return `true`, and some always return `false`. `nil` values always return `nil`.

```
-- These always return true:
yesno('yes')
yesno('y')
yesno('true')
yesno('t')
yesno('1')
yesno(1)
yesno(true)

-- These always return false:
yesno('no')
```



```
yesno('n')
yesno('false')
yesno('f')
yesno('0')
yesno(0)
yesno(false)
```

```
-- A nil value always returns nil:
yesno(nil)
```

String values are converted to lower case before they are matched:

```
-- These always return true:
```

```
yesno('Yes')
yesno('YES')
yesno('yEs')
yesno('Y')
yesno('tRuE')
```

```
-- These always return false:
```

```
yesno('No')
yesno('NO')
yesno('n0')
yesno('N')
yesno('fALsE')
```

Undefined input ('foo')

You can specify a default value if yesno receives input other than that listed above. If you don't supply a default, the module will return `nil` for these inputs.

```
-- These return nil:
```

```
yesno('foo')
yesno({})
yesno(5)
yesno(function() return 'This is a function.' end)
yesno(nil, true)
yesno(nil, 'bar')
```

```
-- These return true:
```

```
yesno('foo', true)
yesno({}, true)
yesno(5, true)
yesno(function() return 'This is a function.' end, true)
```

```
-- These return "bar":
```

```
yesno('foo', 'bar')
yesno({}, 'bar')
yesno(5, 'bar')
yesno(function() return 'This is a function.' end, 'bar')
```

Note that the empty string also functions this way:

```
yesno('') -- Returns nil.
yesno('', true) -- Returns true.
yesno('', 'bar') -- Returns "bar".
```

Although the empty string usually evaluates to false in wikitext, it evaluates to true in Lua. This module prefers the Lua behaviour over the wikitext behaviour. If treating the empty string as false is important for your module, you will need to convert empty strings to a value that evaluates to false before passing them to this module. In the case of arguments received from wikitext, this can be done by using [Module:Arguments](#).

Handling nil results

By definition

```
yesno(nil)           -- Returns nil.
yesno('foo')        -- Returns nil.
yesno(nil, true)     -- Returns nil.
yesno(nil, false)   -- Returns nil.
yesno('foo', true)  -- Returns true.
```

To get the binary true/false-only values, use code like:

```
myvariable = yesno(value) or false -- When value is nil, result is false.
myvariable = yesno(value) or true  -- When value is nil, result is true.
myvariable = yesno('foo') or false -- Unknown string returns nil, result is false.
myvariable = yesno('foo', true) or false -- Default value (here: true) applies,
```

```
-- Function allowing for consistent treatment of boolean-like wikitext input.
-- It works similarly to the template {{yesno}}.

return function (val, default)
    -- If your wiki uses non-ascii characters for any of "yes", "no", etc., y
    -- should replace "val:lower()" with "mw.ustring.lower(val)" in the
    -- following line.
    val = type(val) == 'string' and val:lower() or val
    if val == nil then
        return nil
    elseif val == true
        or val == 'yes'
        or val == 'y'
        or val == 'true'
        or val == 't'
        or val == 'on'
        or tonumber(val) == 1
    then
        return true
    elseif val == false
        or val == 'no'
        or val == 'n'
        or val == 'false'
        or val == 'f'
        or val == 'off'
        or tonumber(val) == 0
```



```
    then
    else      return false
    end      return default
end
```