

# Modul:TableTools

This module includes a number of functions for dealing with Lua tables. It is a meta-module, meant to be called from other Lua modules, and should not be called directly from #invoke.

Inhaltsverzeichnis	
1 Loading the module .....	1
2 isPositiveInteger .....	1
3 isNan .....	2
4 shallowClone .....	2
5 removeDuplicates .....	2
6 numKeys .....	2
7 affixNums .....	2
8 numData .....	3
9 compressSparseArray .....	3
10 sparsepairs .....	3
11 size .....	3
12 keysToList .....	4
13 sortedPairs .....	4
14 isArray .....	4
15 isArrayLike .....	4
16 invert .....	4
17 listToSet .....	4
18 deepCopy .....	5
19 sparseConcat .....	5
20 length .....	5
21 inArray .....	5

## Loading the module

To use any of the functions, first you must load the module.

```
local TableTools = require('Module:TableTools')
```

## isPositiveInteger

```
TableTools.isPositiveInteger(value)
```

Returns `true` if `value` is a positive integer, and `false` if not. Although it doesn't operate on tables, it is included here as it is useful for determining whether a given table key is in the array part or the hash part of a table.

## isNan

```
TableTools.isNan(value)
```

Returns true if *value* is a NaN value, and false if not. Although it doesn't operate on tables, it is included here as it is useful for determining whether a value can be a valid table key. (Lua will generate an error if a NaN value is used as a table key.)

## shallowClone

```
TableTools.shallowClone(t)
```

Returns a clone of a table. The value returned is a new table, but all subtables and functions are shared. Metamethods are respected, but the returned table will have no metatable of its own. If you want to make a new table with no shared subtables and with metatables transferred, you can use [mw.clone](#) instead. If you want to make a new table with no shared subtables and without metatables transferred, use [deepCopy](#) with the `noMetatable` option.

## removeDuplicates

```
TableTools.removeDuplicates(t)
```

Removes duplicate values from an array. This function is only designed to work with standard arrays: keys that are not positive integers are ignored, as are all values after the first nil value. (For arrays containing nil values, you can use [compressSparseArray](#) first.) The function tries to preserve the order of the array: the earliest non-unique value is kept, and all subsequent duplicate values are removed. For example, for the table `Vorlage:Code` `removeDuplicates` will return `Vorlage:Code`.

## numKeys

```
TableTools.numKeys(t)
```

Takes a table *t* and returns an array containing the numbers of any positive integer keys that have non-nil values, sorted in numerical order. For example, for the table `Vorlage:Code`, `numKeys` will return `Vorlage:Code`.

## affixNums

```
TableTools.affixNums(t, prefix, suffix)
```

Takes a table  $t$  and returns an array containing the numbers of keys with the optional prefix *prefix* and the optional suffix *suffix*. For example, for the table `Vorlage:Code` and the prefix `'a'`, `affixNums` will return `Vorlage:Code`. All characters in *prefix* and *suffix* are interpreted literally.

## numData

```
TableTools.numData(t, compress)
```

Given a table with keys like `"foo1"`, `"bar1"`, `"foo2"`, and `"baz2"`, returns a table of subtables in the format `Vorlage:Code`. Keys that don't end with an integer are stored in a subtable named `"other"`. The `compress` option compresses the table so that it can be iterated over with `ipairs`.

## compressSparseArray

```
TableTools.compressSparseArray(t)
```

Takes an array  $t$  with one or more `nil` values, and removes the `nil` values while preserving the order, so that the array can be safely traversed with `ipairs`. Any keys that are not positive integers are removed. For example, for the table `Vorlage:Code`, `compressSparseArray` will return `Vorlage:Code`.

## sparseIpairs

```
TableTools.sparseIpairs(t)
```

This is an iterator function for traversing a sparse array  $t$ . It is similar to `ipairs`, but will continue to iterate until the highest numerical key, whereas `ipairs` may stop after the first `nil` value. Any keys that are not positive integers are ignored.

Usually `sparseIpairs` is used in a generic `for` loop.

```
for i, v in TableTools.sparseIpairs(t) do
  -- code block
end
```

Note that `sparseIpairs` uses the `pairs` function in its implementation. Although some table keys appear to be ignored, all table keys are accessed when it is run.

## size

```
TableTools.size(t)
```



Finds the size of a key/value pair table. For example, for the table `Vorlage:Code`, `size` will return 2. The function will also work on arrays, but for arrays it is more efficient to use the `#` operator. Note that to find the table size, this function uses the `pairs` function to iterate through all of the table keys.

## keysToList

```
TableTools.keysToList(t, keySort, checked)
```

Returns a list of the keys in a table, sorted using either a default comparison function or a custom `keySort` function, which follows the same rules as the `comp` function supplied to `table.sort`. If `keySort` is `false`, no sorting is done. Set `checked` to `true` to skip the internal type checking.

## sortedPairs

```
TableTools.sortedPairs(t, keySort)
```

Iterates through a table, with the keys sorted using the `keysToList` function. If there are only numerical keys, `sparseIpairs` is probably more efficient.

## isArray

```
TableTools.isArray(value)
```

Returns `true` if `value` is a table and all keys are consecutive integers starting at 1.

## isArrayLike

```
TableTools.isArrayLike(value)
```

Returns `true` if `value` is iterable and all keys are consecutive integers starting at 1.

## invert

```
TableTools.invert(arr)
```

Transposes the keys and values in an array. For example, `Vorlage:Code` yields `Vorlage:Code`.

## listToSet

```
TableTools.listToSet(arr)
```



Creates a set from the array part of the table *arr*. Indexing the set by any of the values of the array returns true. For example, `Vorlage:Code` yields `Vorlage:Code`. See also `Module:Lua set` for more advanced ways to create a set.

## deepCopy

```
TableTools.deepCopy(orig, noMetatable, alreadySeen)
```

Creates a copy of the table *orig*. As with `mw.clone`, all values that are not functions are duplicated and the identity of tables is preserved. If *noMetatable* is true, then the metatable (if any) is not copied. Can copy tables loaded with `mw.loadData`.

Similar to `mw.clone`, but `mw.clone` cannot copy tables loaded with `mw.loadData` and does not allow metatables *not* to be copied.

## sparseConcat

```
TableTools.sparseConcat(t, sep, i, j)
```

Concatenates all values in the table that are indexed by a positive integer, in order. For example, `Vorlage:Code` yields `Vorlage:Code` and `Vorlage:Code` yields `Vorlage:Code`.

## length

```
TableTools.length(t, prefix)
```

Finds the length of an array or of a quasi-array with keys with an optional *prefix* such as "data1", "data2", etc. It uses an `exponential search` algorithm to find the length, so as to use as few table lookups as possible.

This algorithm is useful for arrays that use metatables (e.g. `frame.args`) and for quasi-arrays. For normal arrays, just use the `# operator`, as it is implemented in `C` and will be quicker.

## inArray

```
TableTools.inArray(arr, valueToFind)
```

Returns true if *valueToFind* is a member of the array *arr*, and false otherwise.

```
-----  
--                                     TableTools  
--  
-- This module includes a number of functions for dealing with Lua tables.  
-- It is a meta-module, meant to be called from other Lua modules, and should not  
-- be called directly from #invoke.  
-----
```

```
local libraryUtil = require('libraryUtil')

local p = {}

-- Define often-used variables and functions.
local floor = math.floor
local infinity = math.huge
local checkType = libraryUtil.checkType
local checkTypeMulti = libraryUtil.checkTypeMulti

-----
-- isPositiveInteger
--
-- This function returns true if the given value is a positive integer, and false
-- if not. Although it doesn't operate on tables, it is included here as it is
-- useful for determining whether a given table key is in the array part or the
-- hash part of a table.
-----
function p.isPositiveInteger(v)
    return type(v) == 'number' and v >= 1 and floor(v) == v and v < infinity
end

-----
-- isNaN
--
-- This function returns true if the given number is a NaN value, and false if
-- not. Although it doesn't operate on tables, it is included here as it is useful
-- for determining whether a value can be a valid table key. Lua will generate an
-- error if a NaN is used as a table key.
-----
function p.isNaN(v)
    return type(v) == 'number' and v ~= v
end

-----
-- shallowClone
--
-- This returns a clone of a table. The value returned is a new table, but all
-- subtables and functions are shared. Metamethods are respected, but the returned
-- table will have no metatable of its own.
-----
function p.shallowClone(t)
    checkType('shallowClone', 1, t, 'table')
    local ret = {}
    for k, v in pairs(t) do
        ret[k] = v
    end
    return ret
end

-----
-- removeDuplicates
--
-- This removes duplicate values from an array. Non-positive-integer keys are
-- ignored. The earliest value is kept, and all subsequent duplicate values are
-- removed, but otherwise the array order is unchanged.
-----
function p.removeDuplicates(arr)
    checkType('removeDuplicates', 1, arr, 'table')
    local isNaN = p.isNaN
    local ret, exists = {}, {}
    for _, v in ipairs(arr) do
        if isNaN(v) then
```

```
        -- NaNs can't be table keys, and they are also unique, so
        ret[#ret + 1] = v
    else
        if not exists[v] then
            ret[#ret + 1] = v
            exists[v] = true
        end
    end
end
end
return ret
end

-----
-- numKeys
--
-- This takes a table and returns an array containing the numbers of any numerical
-- keys that have non-nil values, sorted in numerical order.
-----
function p.numKeys(t)
    checkType('numKeys', 1, t, 'table')
    local isPositiveInteger = p.isPositiveInteger
    local nums = {}
    for k in pairs(t) do
        if isPositiveInteger(k) then
            nums[#nums + 1] = k
        end
    end
    table.sort(nums)
    return nums
end

-----
-- affixNums
--
-- This takes a table and returns an array containing the numbers of keys with the
-- specified prefix and suffix. For example, for the table
-- {a1 = 'foo', a3 = 'bar', a6 = 'baz'} and the prefix "a", affixNums will return
-- {1, 3, 6}.
-----
function p.affixNums(t, prefix, suffix)
    checkType('affixNums', 1, t, 'table')
    checkType('affixNums', 2, prefix, 'string', true)
    checkType('affixNums', 3, suffix, 'string', true)

    local function cleanPattern(s)
        -- Cleans a pattern so that the magic characters (%).[*+?^$ are
        return s:gsub('([%(%)%%%.%[%]%%*+-%?%^%$])', '%%1')
    end

    prefix = prefix or ''
    suffix = suffix or ''
    prefix = cleanPattern(prefix)
    suffix = cleanPattern(suffix)
    local pattern = '^' .. prefix .. '([1-9]d*)' .. suffix .. '$'

    local nums = {}
    for k in pairs(t) do
        if type(k) == 'string' then
            local num = mw.ustr.match(k, pattern)
            if num then
                nums[#nums + 1] = tonumber(num)
            end
        end
    end
end
```



```
        table.sort(nums)
        return nums
end

-----
-- numData
--
-- Given a table with keys like {"foo1", "bar1", "foo2", "baz2"}, returns a table
-- of subtables in the format
-- {[1] = {foo = 'text', bar = 'text'}, [2] = {foo = 'text', baz = 'text'}}.
-- Keys that don't end with an integer are stored in a subtable named "other". The
-- compress option compresses the table so that it can be iterated over with
-- ipairs.
-----
function p.numData(t, compress)
    checkType('numData', 1, t, 'table')
    checkType('numData', 2, compress, 'boolean', true)
    local ret = {}
    for k, v in pairs(t) do
        local prefix, num = mw.ustr.match(tostring(k), '^([0-9]*)([1-9])')
        if num then
            num = tonumber(num)
            local subtable = ret[num] or {}
            if prefix == '' then
                -- Positional parameters match the blank string;
                prefix = 1
            end
            subtable[prefix] = v
            ret[num] = subtable
        else
            local subtable = ret.other or {}
            subtable[k] = v
            ret.other = subtable
        end
    end
    if compress then
        local other = ret.other
        ret = p.compressSparseArray(ret)
        ret.other = other
    end
    return ret
end

-----
-- compressSparseArray
--
-- This takes an array with one or more nil values, and removes the nil values
-- while preserving the order, so that the array can be safely traversed with
-- ipairs.
-----
function p.compressSparseArray(t)
    checkType('compressSparseArray', 1, t, 'table')
    local ret = {}
    local nums = p.numKeys(t)
    for _, num in ipairs(nums) do
        ret[#ret + 1] = t[num]
    end
    return ret
end

-----
-- sparseIpairs
--
-- This is an iterator for sparse arrays. It can be used like ipairs, but can
```

```
-- handle nil values.
-----
function p.sparsePairs(t)
  checkType('sparsePairs', 1, t, 'table')
  local nums = p.numKeys(t)
  local i = 0
  local lim = #nums
  return function ()
    i = i + 1
    if i <= lim then
      local key = nums[i]
      return key, t[key]
    else
      return nil, nil
    end
  end
end

-----
-- size
--
-- This returns the size of a key/value pair table. It will also work on arrays,
-- but for arrays it is more efficient to use the # operator.
-----
function p.size(t)
  checkType('size', 1, t, 'table')
  local i = 0
  for _ in pairs(t) do
    i = i + 1
  end
  return i
end

local function defaultKeySort(item1, item2)
  -- "number" < "string", so numbers will be sorted before strings.
  local type1, type2 = type(item1), type(item2)
  if type1 ~= type2 then
    return type1 < type2
  elseif type1 == 'table' or type1 == 'boolean' or type1 == 'function' then
    return tostring(item1) < tostring(item2)
  else
    return item1 < item2
  end
end

-----
-- keysToList
--
-- Returns an array of the keys in a table, sorted using either a default
-- comparison function or a custom keySort function.
-----
function p.keysToList(t, keySort, checked)
  if not checked then
    checkType('keysToList', 1, t, 'table')
    checkTypeMulti('keysToList', 2, keySort, {'function', 'boolean',
end

  local arr = {}
  local index = 1
  for k in pairs(t) do
    arr[index] = k
    index = index + 1
  end

  if keySort ~= false then
```

```
        keySort = type(keySort) == 'function' and keySort or defaultKeySort
        table.sort(arr, keySort)
    end
    return arr
end

-----
-- sortedPairs
--
-- Iterates through a table, with the keys sorted using the keysToList function.
-- If there are only numerical keys, sparsePairs is probably more efficient.
-----
function p.sortedPairs(t, keySort)
    checkType('sortedPairs', 1, t, 'table')
    checkType('sortedPairs', 2, keySort, 'function', true)

    local arr = p.keysToList(t, keySort, true)

    local i = 0
    return function ()
        i = i + 1
        local key = arr[i]
        if key ~= nil then
            return key, t[key]
        else
            return nil, nil
        end
    end
end

-----
-- isArray
--
-- Returns true if the given value is a table and all keys are consecutive
-- integers starting at 1.
-----
function p.isArray(v)
    if type(v) ~= 'table' then
        return false
    end
    local i = 0
    for _ in pairs(v) do
        i = i + 1
        if v[i] == nil then
            return false
        end
    end
    return true
end

-----
-- isArrayLike
--
-- Returns true if the given value is iterable and all keys are consecutive
-- integers starting at 1.
-----
function p.isArrayLike(v)
    if not p.call(pairs, v) then
        return false
    end
    local i = 0
    for _ in pairs(v) do
        i = i + 1
    end
end
```

```
        if v[i] == nil then
            return false
        end
    end
    return true
end

-----
-- invert
--
-- Transposes the keys and values in an array. For example, {"a", "b", "c"} ->
-- {a = 1, b = 2, c = 3}. Duplicates are not supported (result values refer to
-- the index of the last duplicate) and NaN values are ignored.
-----
function p.invert(arr)
    checkType("invert", 1, arr, "table")
    local isNan = p.isNan
    local map = {}
    for i, v in ipairs(arr) do
        if not isNan(v) then
            map[v] = i
        end
    end
    return map
end

-----
-- listToSet
--
-- Creates a set from the array part of the table. Indexing the set by any of the
-- values of the array returns true. For example, {"a", "b", "c"} ->
-- {a = true, b = true, c = true}. NaN values are ignored as Lua considers them
-- never equal to any value (including other NaNs or even themselves).
-----
function p.listToSet(arr)
    checkType("listToSet", 1, arr, "table")
    local isNan = p.isNan
    local set = {}
    for _, v in ipairs(arr) do
        if not isNan(v) then
            set[v] = true
        end
    end
    return set
end

-----
-- deepCopy
--
-- Recursive deep copy function. Preserves identities of subtables.
-----
local function _deepCopy(orig, includeMetatable, already_seen)
    -- Stores copies of tables indexed by the original table.
    already_seen = already_seen or {}

    local copy = already_seen[orig]
    if copy ~= nil then
        return copy
    end

    if type(orig) == 'table' then
        copy = {}
    end
end
```



```
        for orig_key, orig_value in pairs(orig) do
            copy[_deepCopy(orig_key, includeMetatable, already_seen)]
        end
        already_seen[orig] = copy

        if includeMetatable then
            local mt = getmetatable(orig)
            if mt ~= nil then
                local mt_copy = _deepCopy(mt, includeMetatable, already_seen)
                setmetatable(copy, mt_copy)
                already_seen[mt] = mt_copy
            end
        end
    else -- number, string, boolean, etc
        copy = orig
    end
    return copy
end

function p.deepCopy(orig, noMetatable, already_seen)
    checkType("deepCopy", 3, already_seen, "table", true)
    return _deepCopy(orig, not noMetatable, already_seen)
end

-----
-- sparseConcat
--
-- Concatenates all values in the table that are indexed by a number, in order.
-- sparseConcat{a, nil, c, d} => "acd"
-- sparseConcat{nil, b, c, d} => "bcd"
-----
function p.sparseConcat(t, sep, i, j)
    local arr = {}

    local arr_i = 0
    for _, v in p.sparsePairs(t) do
        arr_i = arr_i + 1
        arr[arr_i] = v
    end

    return table.concat(arr, sep, i, j)
end

-----
-- length
--
-- Finds the length of an array, or of a quasi-array with keys such as "data1",
-- "data2", etc., using an exponential search algorithm. It is similar to the
-- operator #, but may return a different value when there are gaps in the array
-- portion of the table. Intended to be used on data loaded with mw.loadData. For
-- other tables, use #.
-- Note: #frame.args in frame object always be set to 0, regardless of the number
-- of unnamed template parameters, so use this function for frame.args.
-----
function p.length(t, prefix)
    -- requiring module inline so that [[Module:Exponential search]] which is
    -- only needed by this one function doesn't get millions of transclusions
    local expSearch = require("Module:Exponential search")
    checkType('length', 1, t, 'table')
    checkType('length', 2, prefix, 'string', true)
    return expSearch(function (i)
        local key
        if prefix then
            key = prefix .. tostring(i)
        end
    end)
```



```
                else
                    key = i
                end
                return t[key] ~= nil
            end) or 0
        end

-----
-- inArray
--
-- Returns true if valueToFind is a member of the array, and false otherwise.
-----
function p.inArray(arr, valueToFind)
    checkType("inArray", 1, arr, "table")
    -- if valueToFind is nil, error?

    for _, v in ipairs(arr) do
        if v == valueToFind then
            return true
        end
    end
    return false
end

return p
```