



# Inhaltsverzeichnis

---

# Modul:Math

**Vorlage:For** This module provides a number of mathematical functions. These functions can be used from `#invoke` or from other Lua modules.

Inhaltsverzeichnis	
1 Use from other Lua modules .....	2
2 random .....	2
3 order .....	3
4 precision .....	3
5 max .....	4
6 median .....	4
7 min .....	4
8 sum .....	4
9 average .....	5
10 round .....	5
11 log10 .....	5
12 mod .....	5
13 gcd .....	6
14 precision_format .....	6
15 divide .....	7
16 cleanNumber .....	7
17 See also .....	7

## Use from other Lua modules

To use the module from normal wiki pages, no special preparation is needed. If you are using the module from another Lua module, first you need to load it, like this:

```
local mm = require('Module:Math')
```

(The `mm` variable stands for **Module Math**; you can choose something more descriptive if you prefer.)

Most functions in the module have a version for Lua and a version for `#invoke`. It is possible to use the `#invoke` functions from other Lua modules, but using the Lua functions has the advantage that you do not need to access a Lua [frame object](#). Lua functions are preceded by `_`, whereas `#invoke` functions are not.

## random

**Vorlage:See also**

```
{{#invoke:math|random}}  
{{#invoke:math|random|max_value}}  
{{#invoke:math|random|min_value|max_value}}
```

```
mm._random()  
mm._random(max_value)  
mm._random(min_value, max_value)
```

Generates a random number.

- If no arguments are specified, the number produced is greater than or equal to 0 and less than 1.
- If one argument is provided, the number produced is an integer between 1 and that argument. The argument must be a positive integer.
- If two arguments are provided, the number produced is an integer between the first and second arguments. Both arguments must be integers, but can be negative.

This function will not work properly for numbers less than  $-2^{32}$  and greater than  $2^{32} - 1$ . If you need to use numbers outside of this range, it is recommended that you use [Module:Random](#).

## order

---

```
{{#invoke:math|order|n}}
```

```
mm._order(n)
```

Determines the **order of magnitude** of a number.

## precision

---

```
{{#invoke:math|precision|n}}  
{{#invoke:math|precision|x=n}}
```

```
mm._precision(number_string)
```

Determines the precision of a number. For example, for "4" it will return "0", for "4.567" it will return "3", and for "100" it will return "-2".

The function attempts to parse the string representation of the number, and detects whether the number uses **E notation**. For this reason, when called from Lua, very large numbers or very precise numbers should be directly input as strings to get accurate results. If they are input as numbers, the Lua interpreter will change them to E notation and this function will return the precision of the E notation rather than that of the original number. This is not a problem when the number is called from #invoke, as all input from #invoke is in string format.

## max

```
{#{invoke:math|max|v1|v2|v3|...}}
```

```
mm._max(v1, v2, v3, ...)
```

Returns the maximum value from the values specified. Values that cannot be converted to numbers are ignored.

## median

```
{#{invoke:math|median|v1|v2|v3|...}}
```

```
mm._median(v1, v2, v3, ...)
```

Returns the **median** value from the values specified. Values that cannot be converted to numbers are ignored.

## min

```
{#{invoke:math|min|v1|v2|v3|...}}
```

```
mm._min(v1, v2, v3, ...)
```

Returns the minimum value from the values specified. Values that cannot be converted to numbers are ignored.

## sum

```
{#{invoke:math|sum|v1|v2|v3|...}}
```

```
mm._sum(v1, v2, v3, ...)
```



Returns the sum of the values specified. Values that cannot be converted to numbers are ignored.

## average

```
{{#invoke:math|average|v1|v2|v3|...}}
```

```
mm._average(v1, v2, v3, ...)
```

Returns the average of the values specified. (More precisely, the value returned is the **arithmetic mean**.) Values that cannot be converted to numbers are ignored.

## round

```
{{#invoke:math|round|value|precision}}  
{{#invoke:math|round|value=value|precision=precision}}
```

```
mm._round(value, precision)
```

**Rounds** a number to the specified precision.

Note: As of October 2019, there is a bug in the display of some rounded numbers. When trying to round a number that rounds to "n.0", like "1.02", to the nearest tenth of a digit (i.e. **Vorlage:Para**), this function should display "1.0", but it unexpectedly displays "1". Use the **Vorlage:Para** parameter instead.

## log10

```
{{#invoke:math|log10|x}}
```

```
mm._log10(x)
```

Returns  $\log_{10}(x)$ , the **logarithm** of  $x$  using base 10.

## mod

```
{{#invoke:math|mod|x|y}}
```

```
mm._mod(x, y)
```

Gets  $x$  modulo  $y$ , or the remainder after  $x$  has been divided by  $y$ . This is accurate for integers up to  $2^{53}$ ; for larger integers Lua's modulo operator may return an erroneous value. This function deals with this problem by returning 0 if the modulo given by Lua's modulo operator is less than 0 or greater than  $y$ .

## gcd

```
{#{#invoke:math|gcd|v1|v2|...}}
```

```
mm._gcd(v1, v2, ...)
```

Finds the **greatest common divisor** of the values specified. Values that cannot be converted to numbers are ignored.

## precision\_format

```
{#{#invoke:math|precision_format|value_string|precision}}
```

```
mm._precision_format(value_string, precision)
```

Rounds a number to the specified precision and formats according to rules originally used for **Vorlage:TI**. Output is a string.

Parameter *precision* should be an integer number of digits after the decimal point. Negative values are permitted. Non-integers give unexpected results. Positive values greater than the input precision add zero-padding, negative values greater than the input order can consume all digits.

Formatting 8,765.567 with **Vorlage:TIc** gives:

<i>precision</i>	<b>Result</b>
2	8.765,57
-2	8.800
6	8.765,567000
-6	0
2.5	8.765,568042663 3
-2.5	8.854,377448471 5

## divide

```
{#{#invoke:Math|divide|x|y|round=|precision=}}
```

```
mm._divide(x, y, round, precision)
```

Divide x by y.

- If y is not a number, it is returned.
- Otherwise, if x is not a number, it is returned.
- If round is true ("yes" for #invoke), the result has no decimals
- Precision indicates how many digits of precision the result should have

If any of the arguments contain HTML tags, they are returned unchanged, allowing any errors in calculating the arguments to the division function to be propagated to the calling template.

## cleanNumber

```
local number, number_string = mm._cleanNumber(number_string)
```

A helper function that can be called from other Lua modules, but not from #invoke. This takes a string or a number value as input, and if the value can be converted to a number, cleanNumber returns the number and the number string. If the value cannot be converted to a number, cleanNumber returns `nil, nil`.

## See also

[Vorlage:Math templates](#)

```
--[[
This module provides a number of basic mathematical operations.
]]
local yesno, getArgs -- lazily initialized
local p = {} -- Holds functions to be returned from #invoke, and functions to make
local wrap = {} -- Holds wrapper functions that process arguments from #invoke.

--[[
Helper functions used to avoid redundant code.
]]
local function err(msg)
    -- Generates wikitext error messages.
    return mw.ustext.format('<strong class="error">Formatting error: %s</strong>')
end
```



```
local function unpackNumberArgs(args)
    -- Returns an unpacked list of arguments specified with numerical keys.
    local ret = {}
    for k, v in pairs(args) do
        if type(k) == 'number' then
            table.insert(ret, v)
        end
    end
    return unpack(ret)
end

local function makeArgArray(...)
    -- Makes an array of arguments from a list of arguments that might include nils or non-numbers.
    local args = {...} -- Table of arguments. It might contain nils or non-numbers.
    local nums = {} -- Stores the numbers of valid numerical arguments.
    local ret = {}
    for k, v in pairs(args) do
        v = p._cleanNumber(v)
        if v then
            nums[#nums + 1] = k
            args[k] = v
        end
    end
    table.sort(nums)
    for i, num in ipairs(nums) do
        ret[#ret + 1] = args[num]
    end
    return ret
end

local function fold(func, ...)
    -- Use a function on all supplied arguments, and return the result. The function should
    -- and must return a number as an output. This number is then supplied as the first argument
    local vals = makeArgArray(...)
    local count = #vals -- The number of valid arguments
    if count == 0 then return
        -- Exit if we have no valid args, otherwise removing the first argument
        nil, 0
    end
    local ret = table.remove(vals, 1)
    for _, val in ipairs(vals) do
        ret = func(ret, val)
    end
    return ret, count
end

--[[
Fold arguments by selectively choosing values (func should return when to choose
]]
local function binary_fold(func, ...)
    local value = fold((function(a, b) if func(a, b) then return a else return b end))
    return value
end

--[[
random

Generate a random number

Usage:
{{#invoke: Math | random }}
{{#invoke: Math | random | maximum value }}
{{#invoke: Math | random | minimum value | maximum value }}
]]
```

```
function wrap.random(args)
    local first = p._cleanNumber(args[1])
    local second = p._cleanNumber(args[2])
    return p._random(first, second)
end

function p._random(first, second)
    math.randomseed(mw.site.stats.edits + mw.site.stats.pages + os.time() + math.random())
    -- math.random will throw an error if given an explicit nil parameter, so we check for it
    if first and second then
        if first <= second then -- math.random doesn't allow the first number to be greater than the second
            return math.random(first, second)
        end
    elseif first then
        return math.random(first)
    else
        return math.random()
    end
end

--[[
order

Determine order of magnitude of a number

Usage:
{{#invoke: Math | order | value }}
]]

function wrap.order(args)
    local input_string = (args[1] or args.x or '0');
    local input_number = p._cleanNumber(input_string);
    if input_number == nil then
        return err('order of magnitude input appears non-numeric')
    else
        return p._order(input_number)
    end
end

function p._order(x)
    if x == 0 then return 0 end
    return math.floor(math.log10(math.abs(x)))
end

--[[
precision

Determines the precision of a number using the string representation

Usage:
{{ #invoke: Math | precision | value }}
]]

function wrap.precision(args)
    local input_string = (args[1] or args.x or '0');
    local trap_fraction = args.check_fraction;
    local input_number;

    if not yesno then
        yesno = require('Module:Yesno')
    end
    if yesno(trap_fraction, true) then -- Returns true for all input except for the string representation of a fraction
        local pos = string.find(input_string, '/', 1, true);
```

```
        if pos ~= nil then
            if string.find(input_string, '/', pos + 1, true) == nil then
                local denominator = string.sub(input_string, pos + 1, string.len(input_string))
                local denom_value = tonumber(denominator)
                if denom_value ~= nil then
                    return math.log10(denom_value)
                end
            end
        end
    end

    input_number, input_string = p._cleanNumber(input_string)
    if input_string == nil then
        return err('precision input appears non-numeric')
    else
        return p._precision(input_string)
    end
end

function p._precision(x)
    if type(x) == 'number' then
        x = tostring(x)
    end
    x = string.upper(x)

    local decimal = x:find('%.')
    local exponent_pos = x:find('E')
    local result = 0

    if exponent_pos ~= nil then
        local exponent = string.sub(x, exponent_pos + 1, string.len(x))
        x = string.sub(x, 1, exponent_pos - 1)
        result = result - tonumber(exponent)
    end

    if decimal ~= nil then
        result = result + string.len(x) - decimal
    end

    return result
end

local pos = string.len(x)
while x:byte(pos) == string.byte('0') do
    pos = pos - 1
    result = result - 1
    if pos <= 0 then
        return 0
    end
end

return result
end

--[[
max

Finds the maximum argument

Usage:
{#{invoke:Math| max | value1 | value2 | ... }}

Note, any values that do not evaluate to numbers are ignored.
]]
```



```
function wrap.max(args)
    return p._max(unpackNumberArgs(args))
end

function p._max(...)
    local max_value = binary_fold((function(a, b) return a > b end), ...)
    if max_value then
        return max_value
    end
end

--[[
median

Find the median of set of numbers

Usage:
{#{invoke:Math | median | number1 | number2 | ...}}
OR
{#{invoke:Math | median }}
]]

function wrap.median(args)
    return p._median(unpackNumberArgs(args))
end

function p._median(...)
    local vals = makeArgArray(...)
    local count = #vals
    table.sort(vals)

    if count == 0 then
        return 0
    end

    if p._mod(count, 2) == 0 then
        return (vals[count/2] + vals[count/2+1])/2
    else
        return vals[math.ceil(count/2)]
    end
end

end

--[[
min

Finds the minimum argument

Usage:
{#{invoke:Math| min | value1 | value2 | ... }}
OR
{#{invoke:Math| min }}

When used with no arguments, it takes its input from the parent
frame. Note, any values that do not evaluate to numbers are ignored.
]]

function wrap.min(args)
    return p._min(unpackNumberArgs(args))
end

function p._min(...)
    local min_value = binary_fold((function(a, b) return a < b end), ...)
    if min_value then
        return min_value
    end
end
```

```
end
end

--[
sum

Finds the sum

Usage:
{#{invoke:Math| sum | value1 | value2 | ... }}
OR
{#{invoke:Math| sum }}

Note, any values that do not evaluate to numbers are ignored.
]]

function wrap.sum(args)
  return p._sum(unpackNumberArgs(args))
end

function p._sum(...)
  local sums, count = fold((function(a, b) return a + b end), ...)
  if not sums then
    return 0
  else
    return sums
  end
end

end

--[
average

Finds the average

Usage:
{#{invoke:Math| average | value1 | value2 | ... }}
OR
{#{invoke:Math| average }}

Note, any values that do not evaluate to numbers are ignored.
]]

function wrap.average(args)
  return p._average(unpackNumberArgs(args))
end

function p._average(...)
  local sum, count = fold((function(a, b) return a + b end), ...)
  if not sum then
    return 0
  else
    return sum / count
  end
end

end

--[
round

Rounds a number to specified precision

Usage:
{#{invoke:Math| round | value | precision }}

--]]
```



```
function wrap.round(args)
  local value = p._cleanNumber(args[1] or args.value or 0)
  local precision = p._cleanNumber(args[2] or args.precision or 0)
  if value == nil or precision == nil then
    return err('round input appears non-numeric')
  else
    return p._round(value, precision)
  end
end
```

```
function p._round(value, precision)
  local rescale = math.pow(10, precision or 0);
  return math.floor(value * rescale + 0.5) / rescale;
end
```

```
--[[
log10
```

returns the log (base 10) of a number

```
Usage:
{#{invoke:Math | log10 | x }}
]]
```

```
function wrap.log10(args)
  return math.log10(args[1])
end
```

```
--[[
mod
```

Implements the modulo operator

```
Usage:
{#{invoke:Math | mod | x | y }}
--]]
```

```
function wrap.mod(args)
  local x = p._cleanNumber(args[1])
  local y = p._cleanNumber(args[2])
  if not x then
    return err('first argument to mod appears non-numeric')
  elseif not y then
    return err('second argument to mod appears non-numeric')
  else
    return p._mod(x, y)
  end
end
```

```
function p._mod(x, y)
  local ret = x % y
  if not (0 <= ret and ret < y) then
    ret = 0
  end
  return ret
end
```

```
--[[
gcd
```

Calculates the greatest common divisor of multiple numbers

```
Usage:
{{#invoke:Math | gcd | value 1 | value 2 | value 3 | ... }}
--]]
```

```
function wrap.gcd(args)
    return p._gcd(unpackNumberArgs(args))
end

function p._gcd(...)
    local function findGcd(a, b)
        local r = b
        local oldr = a
        while r ~= 0 do
            local quotient = math.floor(oldr / r)
            oldr, r = r, oldr - quotient * r
        end
        if oldr < 0 then
            oldr = oldr * -1
        end
        return oldr
    end
    local result, count = fold(findGcd, ...)
    return result
end
```

```
--[[
precision_format
```

Rounds a number to the specified precision and formats according to rules originally used for `mw.Rnd`. Output is a string.

```
Usage:
{{#invoke: Math | precision_format | number | precision }}
]]
```

```
function wrap.precision_format(args)
    local value_string = args[1] or 0
    local precision = args[2] or 0
    return p._precision_format(value_string, precision)
end
```

```
function p._precision_format(value_string, precision)
    -- For access to Mediawiki built-in formatter.
    local lang = mw.getContentLanguage();

    local value
    value, value_string = p._cleanNumber(value_string)
    precision = p._cleanNumber(precision)

    -- Check for non-numeric input
    if value == nil or precision == nil then
        return err('invalid input when rounding')
    end

    local current_precision = p._precision(value)
    local order = p._order(value)

    -- Due to round-off effects it is necessary to limit the returned precision
    -- some circumstances because the terminal digits will be inaccurately rounded
    if order + precision >= 14 then
        if order + p._precision(value_string) >= 14 then
            precision = 13 - order;
        end
    end

end
```

```
-- If rounding off, truncate extra digits
if precision < current_precision then
    value = p._round(value, precision)
    current_precision = p._precision(value)
end

local formatted_num = lang:formatNum(math.abs(value))
local sign

-- Use proper unary minus sign rather than ASCII default
if value < 0 then
    sign = '-'
else
    sign = ''
end

-- Handle cases requiring scientific notation
if string.find(formatted_num, 'E', 1, true) ~= nil or math.abs(order) >=
    value = value * math.pow(10, -order)
    current_precision = current_precision + order
    precision = precision + order
    formatted_num = lang:formatNum(math.abs(value))
else
    order = 0;
end
formatted_num = sign .. formatted_num

-- Pad with zeros, if needed
if current_precision < precision then
    local padding
    if current_precision <= 0 then
        if precision > 0 then
            local zero_sep = lang:formatNum(1.1)
            formatted_num = formatted_num .. zero_sep:sub(2,2)

            padding = precision
            if padding > 20 then
                padding = 20
            end

            formatted_num = formatted_num .. string.rep('0',
                end
        else
            padding = precision - current_precision
            if padding > 20 then
                padding = 20
            end
            formatted_num = formatted_num .. string.rep('0', padding)
        end
    end
end

-- Add exponential notation, if necessary.
if order ~= 0 then
    -- Use proper unary minus sign rather than ASCII default
    if order < 0 then
        order = '-' .. lang:formatNum(math.abs(order))
    else
        order = lang:formatNum(order)
    end

    formatted_num = formatted_num .. 'Ausgabe: 27.06.2026
```



```
        return formatted_num
    end

    --[[
    divide

    Implements the division operator

    Usage:
    {{#invoke:Math | divide | x | y | round= | precision= }}

    --]]
    function wrap.divide(args)
        local x = args[1]
        local y = args[2]
        local round = args.round
        local precision = args.precision
        if not yesno then
            yesno = require('Module:Yesno')
        end
        return p._divide(x, y, yesno(round), precision)
    end

    function p._divide(x, y, round, precision)
        if y == nil or y == "" then
            return err("Empty divisor")
        elseif not tonumber(y) then
            if type(y) == 'string' and string.sub(y, 1, 1) == '<' then
                return y
            else
                return err("Not a number: " .. y)
            end
        elseif x == nil or x == "" then
            return err("Empty dividend")
        elseif not tonumber(x) then
            if type(x) == 'string' and string.sub(x, 1, 1) == '<' then
                return x
            else
                return err("Not a number: " .. x)
            end
        else
            local z = x / y
            if round then
                return p._round(z, 0)
            elseif precision then
                return p._round(z, precision)
            else
                return z
            end
        end
    end

end

--[[
Helper function that interprets the input numerically. If the
input does not appear to be a number, attempts evaluating it as
a parser functions expression.
]]

function p._cleanNumber(number_string)
    if type(number_string) == 'number' then
        -- We were passed a number, so we don't need to do any processing
        return number_string, tostring(number_string)
    elseif type(number_string) ~= 'string' or not number_string:find('%S') th
        -- We were passed a non-string or a blank string, so exit.
```

```
        return nil, nil;
    end

    -- Attempt basic conversion
    local number = tonumber(number_string)

    -- If failed, attempt to evaluate input as an expression
    if number == nil then
        local success, result = pcall(mw.ext.ParserFunctions.expr, number)
        if success then
            number = tonumber(result)
            number_string = tostring(number)
        else
            number = nil
            number_string = nil
        end
    else
        number_string = number_string:match("^%s*(.)%s*$") -- String is
        number_string = number_string:match("^%+(.*)$") or number_string
        if number_string:find('^%-?0[xX]') then
            -- Number is using 0xnnn notation to indicate base 16; us
            number_string = tostring(number)
        end
    end

    return number, number_string
end

--[[
Wrapper function that does basic argument processing. This ensures that all funct
frame or the parent frame, and it also trims whitespace for all arguments and ren
]]

local mt = { __index = function(t, k)
    return function(frame)
        if not getArgs then
            getArgs = require('Module:Arguments').getArgs
        end
        return wrap[k](getArgs(frame)) -- Argument processing is left to
    end
end }

return setmetatable(p, mt)
```