



# Inhaltsverzeichnis

---

# Modul:Gapnum

---

This module is used by [Vorlage:TI](#).

## Use in other modules

---

### gaps

---

The `gaps` function can be useful for **formatting** in other modules that work with displaying large numbers.

```
local gaps = require('Module:Gapnum').gaps
```

Using the `gaps` function, the first argument is the number to format. The second argument can be a table with keys that tell the module how to format. The table keys that can be used are:

- `gap` - a number with CSS units (px, em, en, etc) that define the size of the gap between numbers. If blank, the module will use `0.25em`.
- `prec` - a number that determines the precision of the decimal part of the number. If the precision is less than the number of digits, extra digits will be removed without rounding; if it is more, zeroes will be added to the end to create the desired precision. If blank, the module will use `-1`, which means the precision will be the same as the number given; no digits added or removed.

Note that the return statement is a table. This means more styling or text can be added to the wrapper span tag, but it may also mean that `tostring()` may be required when used in other modules.

```
local gaps = require('Module:Gapnum').gaps

function example()
local n = 123456.78900011
-- Example for just simple formatting of a number
-- n_gaps will use the default, .25em gaps and no change in precision
-- The result will have its gaps created with inline css
-- But the result would look like:
-- 123 456.789 000 11
local n_gaps = gaps(n)

-- Different gap size
-- These will format n into the same groups as above
-- But the spaces between the groups will be larger and smaller, respectively
local n_big_gaps = gaps(n, {gap='1em'})
local n_small_gaps = gaps(n, {gap='1px'})

-- Different precision
-- n_prec_5 will use the number 123456.78900
-- The result would look like:
-- 123 456.789 00
local n_prec_5 = gaps(n, {prec=5})
-- n_prec_10 will use the number 123456.7890001100
```



```
-- The result would look like:  
-- 123 456.789 000 1100  
local n_prec_10 = gaps(n, {prec=10})  
  
-- Both different gaps and precision can be used:  
local n_big_5 = gaps(n, {gap='1em', prec=5})  
local n_small_10 = gaps(n, {gap='1px', prec=10})  
end
```

## groups

The `groups` function can be used in other modules to separate a number into groups as `gaps` does, but instead of a formatted string, the function will return tables whose elements are the separated groups.

```
local groups = require('Module:Gapnum').groups  
  
function example()  
-- This will return one table:  
-- {123,456}  
local n1 = groups(123456)  
  
-- This will return two tables, each assigned to a different variable:  
-- n2a will be:  
-- {1,234}  
-- n2b will be:  
-- {567,89}  
local n2a,n2b = groups(1234.56789)  
  
-- This will return two tables:  
-- An integer part is always returned, even if it is 0  
-- n3a will be:  
-- {0}  
-- n3b will be:  
-- {123,4567}  
local n3a,n3b = groups(0.1234567)  
  
-- Just like the other functions, a precision can be defined  
-- precision is simply the second parameter  
-- n4a will be:  
-- {123}  
-- n4b will be:  
-- {456,700,00}  
local n4a,n4b = groups(123.4567,8)  
end
```

```
local p = {}  
  
local getArgs  
  
function p.main(frame)  
    if not getArgs then  
        getArgs = require('Module:Arguments').getArgs  
    end  
  
    local args = getArgs(frame, {wrappers = 'Template:Gapnum'})  
    local n = args[1]
```



```
    if not n then
        error('Parameter 1 is required')
    elseif not tonumber(n) and not tonumber(n, 36) then -- Validates any number
        error('Unable to convert "' .. args[1] .. '" to a number')
    end

    local gap = args.gap
    local precision = tonumber(args.prec)

    return p.gaps(n, {gap=gap, prec=precision})
end

-- Not named p._main so that it has a better function name when required by Module
function p.gaps(n, tbl)
    local nstr = tostring(n)
    if not tbl then
        tbl = {}
    end
    local gap = tbl.gap or '.25em'

    local int_part, frac_part = p.groups(n, tbl.prec)

    local ret = mw.html.create('span')
                                                :css('white-space', 'nowrap')
                                                -- No gap necessary on first part
                                                :wikitext(table.remove(tbl, 1))

    -- Build int part
    for _, v in ipairs(int_part) do
        ret:tag('span')
            :css('margin-left', gap)
            :wikitext(v)
    end

    if frac_part then
        -- The first group after the decimal shouldn't have a gap
        ret:wikitext('.') .. table.remove(frac_part, 1)
        -- Build frac part
        for _, v in ipairs(frac_part) do
            ret:tag('span')
                :css('margin-left', gap)
                :wikitext(v)
        end
    end

    return ret
end

-- Creates tables where each element is a different group of the number
function p.groups(num, precision)
    local nstr = tostring(num)
    if not precision then
        precision = -1
    end

    local decimalloc = nstr:find('.', 1, true)
    local int_part, frac_part
    if decimalloc == nil then
        int_part = nstr
    else
        int_part = nstr:sub(1, decimalloc-1)
        frac_part = nstr:sub(decimalloc + 1)
    end

    -- only define ret_i as an empty table, let ret_d stay nil
end
```

```
local ret_i,ret_d = {}
-- Loop to handle most of the groupings; from right to left, so that if a
while int_part:len() > 3 do
  -- Insert in first spot, since we're moving backwards
  table.insert(ret_i,1,int_part:sub(-3))
  int_part = int_part:sub(1,-4)
end
-- handle any left over numbers
if int_part:len() > 0 then
  table.insert(ret_i,1,int_part)
end

if precision ~= 0 and frac_part then
  ret_d = {}
  if precision == -1 then
    precision = frac_part:len()
  end
  -- Reduce the length of the string if required precision is less
  -- OR
  -- Increase it (by adding 0s) if the required precision is more t
  local offset = precision - frac_part:len()
  if offset < 0 then
    frac_part = frac_part:sub(1,precision)
  elseif offset > 0 then
    frac_part = frac_part .. string.rep('0', offset)
  end

  -- Allow groups of 3 or 2 (3 first)
  for v in string.gmatch(frac_part,'%d%d%d?') do
    table.insert(ret_d,v)
  end
  -- Preference for groups of 4 instead of groups of 1 at the end
  if #frac_part % 3 == 1 then
    if frac_part:len() == 1 then
      ret_d = {frac_part}
    else
      local last_g = ret_d[#ret_d] or ''
      last_g = last_g..frac_part:sub(-1)
      ret_d[#ret_d] = last_g
    end
  end
end

return ret_i,ret_d
end

return p
```