Module:Math

Wechseln zu:Navigation, Suche

WikipediaEN:Module:Math

```lua
--[[

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 available to other Lua modules.
local wrap = {} -- Holds wrapper functions that process arguments from
#invoke. These act as intemediary between functions meant for #invoke and
functions meant for Lua.

--[[
Helper functions used to avoid redundant code.
]]

local function err(msg)
	-- Generates wikitext error messages.
	return mw.ustring.format('<strong class="error">Formatting error:
%s</strong>', msg)
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.
	local args = {...} -- Table of arguments. It might contain nils
or non-number values, so we can't use ipairs.
	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 must accept two numbers as parameters,
        -- and must return a number as an output. This number is then
supplied as input to the next function call.
        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 arg would cause an error.
                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 the current "dominant" value).
]]
local function binary_fold(func, ...)
        local value = fold((function(a, b) if func(a, b) then return a
else return b end 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.floor(os.clock() * 1000000000))
	-- math.random will throw an error if given an explicit nil
parameter, so we need to use if statements to check the params.
	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
```

```
Detemines 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 nil, false, "no", "n", "0" and a few others. See [[Module:Yesno]].
		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, -1);
				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)
                x = string.sub(x, 1, exponent_pos - 1)
                result = result - tonumber(exponent)
        end

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

--[[
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

--[[
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

--[[
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

--[[
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 }}
```

```lua
--]]

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
```

```lua
end

--[[
precision_format

Rounds a number to the specified precision and formats according to rules
originally used for {{template: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 neccesary to limit the returned
precision under
	-- some circumstances because the terminal digits will be
inaccurately reported.
	if order + precision >= 14 then
		orig_precision = p._precision(value_string)
		if order + orig_precision >= 14 then
			precision = 13 - order;
		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) >= 9 then
                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', padding)
                        end
                else
                        padding = precision - current_precision
                        if padding > 20 then
                                padding = 20
                        end
                        formatted_num = formatted_num .. string.rep('0',
padding)
                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 .. '<span style="margin:0
.15em 0 .25em">×</span>10<sup>' .. order .. '</sup>'
        end

        return formatted_num
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') then
                -- 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_string)
                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 valid but may contain padding, clean it.
```

```
                number_string = number_string:match("^%+(.*)$") or
number_string -- Trim any leading + signs.
                if number_string:find('^%-?0[xX]') then
                        -- Number is using 0xnnn notation to indicate
base 16; use the number that Lua detected instead.
                        number_string = tostring(number)
                end
        end

        return number, number_string
end

--[[
Wrapper function that does basic argument processing. This ensures that
all functions from #invoke can use either the current
frame or the parent frame, and it also trims whitespace for all arguments
and removes blank arguments.
]]

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 Module:Arguments. Whitespace is trimmed and blank arguments are
removed.
        end
end }

return setmetatable(p, mt)
```