Search (S)

Overview
Tutorial
Reference
LANGUAGE
Syntax
Styling
Scripting
Context
LIBRARY
Foundations
Arguments
Array
Assert
Auto
Boolean
Bytes
Calculation
Content
Datetime
Dictionary
Duration
Evaluate
Float
Function
Integer
Label
Module

> Reference > Foundations > Calculation

Calculation

Module for calculations and processing of numeric values.

These definitions are part of the `calc` module and not imported by default. In addition to the functions listed below, the `calc` module also defines the constants `pi`, `tau`, `e`, `inf`, and `nan`.

Functions

`abs`

Calculates the absolute value of a numeric value.

```
calc.abs( int float length angle ratio fraction
```

```
#calc.abs(-5) \
#calc.abs(5pt - 2cm) \
#calc.abs(2fr)
```

```
5
51.69pt
2fr
```

`value`

`int` or `float` or `length` or `angle` or `ratio` or `fraction` *Required* *Positional*

The value whose absolute value to calculate.

pow

Raises a value to some exponent.

```
calc.pow(
  int float,
  int float,
) -> int float
```

```
#calc.pow(2, 3)
```

```
8
```

base int or float *Required* *Positional*

The base of the power.

exponent int or float *Required* *Positional*

The exponent of the power.

exp

Raises a value to some exponent of e.

```
calc.exp( int float ) -> float
```

```
#calc.exp(1)
```

```
2.718281828459045
```

exponent int or float *Required* *Positional*

The exponent of the power.

sqrt

Calculates the square root of a number.

```
calc.sqrt( int float ) -> float
```

```
#calc.sqrt(16) \
#calc.sqrt(2.5)
```

```
4
1.5811388300841898
```

value `int` or `float` *Required* *Positional*

The number whose square root to calculate. Must be non-negative.

root

Calculates the real nth root of a number.

If the number is negative, then n must be odd.

```
calc.root(
  float,
  int,
) -> float
```

```
#calc.root(16.0, 4) \
#calc.root(27.0, 3)
```

```
2
3
```

radicand `float` *Required* *Positional*

The expression to take the root of

index `int` *Required* *Positional*

Which root of the radicand to take

`sin`

Calculates the sine of an angle.

When called with an integer or a float, they will be interpreted as radians.

```
calc.sin( int float angle ) -> float
```

```
#assert(calc.sin(90deg) == calc.sin(-270deg))
#calc.sin(1.5) \
#calc.sin(90deg)
```

```
0.9974949866040544
1
```

angle `int` or `float` or `angle` *Required* *Positional*

The angle whose sine to calculate.

`cos`

Calculates the cosine of an angle.

When called with an integer or a float, they will be interpreted as radians.

```
calc.cos( int float angle ) -> float
```

```
#calc.cos(90deg) \
#calc.cos(1.5) \
#calc.cos(90deg)
```

```
0.00000000000000006123233995736766
0.0707372016677029
0.00000000000000006123233995736766
```

angle `int` or `float` or `angle` *Required* *Positional*

The angle whose cosine to calculate.

tan

Calculates the tangent of an angle.

When called with an integer or a float, they will be interpreted as radians.

```
calc.tan( int float angle ) -> float
```

```
#calc.tan(1.5) \
#calc.tan(90deg)
```

```
14.101419947171719
16331239353195370
```

angle `int` or `float` or `angle` *Required* *Positional*

The angle whose tangent to calculate.

asin

Calculates the arcsine of a number.

```
calc.asin( int float ) -> angle
```

```
#calc.asin(0) \
#calc.asin(1)
```

```
0deg
90deg
```

value `int` or `float` *Required* *Positional*

The number whose arcsine to calculate. Must be between -1 and 1.

acos

Calculates the arccosine of a number.

```
calc.acos( int float ) -> angle
```

```
#calc.acos(0) \
#calc.acos(1)
```

```
90deg
0deg
```

value int or float *Required* *Positional*

The number whose arcsine to calculate. Must be between -1 and 1.

atan

Calculates the arctangent of a number.

```
calc.atan( int float ) -> angle
```

```
#calc.atan(0) \
#calc.atan(1)
```

```
0deg
45deg
```

value int or float *Required* *Positional*

The number whose arctangent to calculate.

atan2

Calculates the four-quadrant arctangent of a coordinate.

The arguments are `(x, y)`, not `(y, x)`.

```
calc.atan2(
  int float,
  int float,
) -> angle
```

```
#calc.atan2(1, 1) \
#calc.atan2(-2, -3)
```

```
45deg
-123.69deg
```

x `int` or `float` *Required* *Positional*

The X coordinate.

y `int` or `float` *Required* *Positional*

The Y coordinate.

sinh

Calculates the hyperbolic sine of a hyperbolic angle.

```
calc.sinh(float) -> float
```

```
#calc.sinh(0) \
#calc.sinh(1.5)
```

```
0
2.1292794550948173
```

value `float` *Required* *Positional*

The hyperbolic angle whose hyperbolic sine to calculate.

cosh

Calculates the hyperbolic cosine of a hyperbolic angle.

```
calc.cosh( float ) -> float
```

```
#calc.cosh(0) \
#calc.cosh(1.5)
```

```
1
2.352409615243247
```

value `float` *Required* *Positional*

The hyperbolic angle whose hyperbolic cosine to calculate.

tanh

Calculates the hyperbolic tangent of an hyperbolic angle.

```
calc.tanh( float ) -> float
```

```
#calc.tanh(0) \
#calc.tanh(1.5)
```

```
0
0.9051482536448664
```

value `float` *Required* *Positional*

The hyperbolic angle whose hyperbolic tangent to calculate.

`log`

Calculates the logarithm of a number.

If the base is not specified, the logarithm is calculated in base 10.

```
calc.log(
  int float,
  base: float,
) -> float
```

```
#calc.log(100)
```

```
2
```

`value` `int` or `float` *Required* *Positional*

The number whose logarithm to calculate. Must be strictly positive.

`base` `float`

The base of the logarithm. May not be zero.

Default: `10.0`

`ln`

Calculates the natural logarithm of a number.

```
calc.ln(int float) -> float
```

```
#calc.ln(calc.e)
```

```
1
```

`value` `int` or `float` *Required* *Positional*

The number whose logarithm to calculate. Must be strictly positive.

`fact`

Calculates the factorial of a number.

```
calc.fact(int) -> int
```

```
#calc.fact(5)
```

```
120
```

`number` `int` *Required* *Positional*

The number whose factorial to calculate. Must be non-negative.

`perm`

Calculates a permutation.

Returns the `k`-permutation of `n`, or the number of ways to choose `k` items from a set of `n` with regard to order.

```
calc.perm(
  int,
  int,
) -> int
```

```
$ "perm"(n, k) &= n!/((n - k)!) \
  "perm"(5, 3) &= #calc.perm(5, 3) $
```

$$\text{perm}(n,k) = \frac{n!}{(n-k)!}$$

$$\text{perm}(5,3) = 60$$

base `int` *Required* *Positional*

The base number. Must be non-negative.

numbers `int` *Required* *Positional*

The number of permutations. Must be non-negative.

binom

Calculates a binomial coefficient.

Returns the `k`-combination of `n`, or the number of ways to choose `k` items from a set of `n` without regard to order.

```
calc.binom(
  int,
  int,
) -> int
```

```
#calc.binom(10, 5)
```

```
252
```

n `int` *Required* *Positional*

The upper coefficient. Must be non-negative.

k `int` *Required* *Positional*

The lower coefficient. Must be non-negative.

gcd

Calculates the greatest common divisor of two integers.

```
calc.gcd(
  int,
  int,
) -> int
```

```
#calc.gcd(7, 42)
```

```
7
```

a `int` *Required* *Positional*

The first integer.

b `int` *Required* *Positional*

The second integer.

`lcm`

Calculates the least common multiple of two integers.

```
calc.lcm(
  int,
  int,
) -> int
```

```
#calc.lcm(96, 13)
```

```
1248
```

a `int` *Required* *Positional*

The first integer.

b `int` *Required* *Positional*

The second integer.

`floor`

Rounds a number down to the nearest integer.

If the number is already an integer, it is returned unchanged.

```
calc.floor( int float ) -> int
```

```
#assert(calc.floor(3.14) == 3)
#assert(calc.floor(3) == 3)
#calc.floor(500.1)
```

```
500
```

value `int` or `float` *Required* *Positional*

The number to round down.

ceil

Rounds a number up to the nearest integer.

If the number is already an integer, it is returned unchanged.

```
calc.ceil( int float ) -> int
```

```
#assert(calc.ceil(3.14) == 4)
#assert(calc.ceil(3) == 3)
#calc.ceil(500.1)
```

```
501
```

value `int` or `float` *Required* *Positional*

The number to round up.

trunc

Returns the integer part of a number.

If the number is already an integer, it is returned unchanged.

```
calc.trunc( int float ) -> int
```

```
#assert(calc.trunc(3) == 3)
#assert(calc.trunc(-3.7) == -3)
#calc.trunc(15.9)
```

```
15
```

value `int` or `float` *Required* *Positional*

The number to truncate.

fract

Returns the fractional part of a number.

If the number is an integer, returns `0`.

```
calc.fract( int float ) -> int float
```

```
#assert(calc.fract(3) == 0)
#calc.fract(-3.1)
```

```
−0.10000000000000009
```

value `int` or `float` *Required* *Positional*

The number to truncate.

round

Rounds a number to the nearest integer.

Optionally, a number of decimal places can be specified.

```
calc.round(
  int float,
  digits: int,
) -> int float
```

```
#assert(calc.round(3.14) == 3)
#assert(calc.round(3.5) == 4)
#calc.round(3.1415, digits: 2)
```

```
3.14
```

value `int` or `float` *Required* *Positional*

The number to round.

digits `int`

The number of decimal places.

Default: `0`

clamp

Clamps a number between a minimum and maximum value.

```
calc.clamp(
  int float,
  int float,
  int float,
) -> int float
```

```
#assert(calc.clamp(5, 0, 10) == 5)
#assert(calc.clamp(5, 6, 10) == 6)
#calc.clamp(5, 0, 4)
```

```
4
```

value `int` or `float` *Required* *Positional*

The number to clamp.

min `int` or `float` *Required* *Positional*

The inclusive minimum value.

max `int` or `float` *Required* *Positional*

The inclusive maximum value.

min

Determines the minimum of a sequence of values.

```
calc.min(.. any ) -> any
```

```
#calc.min(1, -3, -5, 20, 3, 6) \
#calc.min("typst", "in", "beta")
```

```
−5
beta
```

values `any` *Required* *Positional* *Variadic*

The sequence of values from which to extract the minimum. Must not be empty.

max

Determines the maximum of a sequence of values.

```
calc.max(.. any ) -> any
```

```
#calc.max(1, -3, -5, 20, 3, 6) \
#calc.max("typst", "in", "beta")
```

```
20
```

```
20
typst
```

values `any` *Required* *Positional* *Variadic*

The sequence of values from which to extract the maximum. Must not be empty.

even

Determines whether an integer is even.

```
calc.even( int ) -> bool
```

```
#calc.even(4) \
#calc.even(5) \
#range(10).filter(calc.even)
```

```
true
false
(0, 2, 4, 6, 8)
```

value `int` *Required* *Positional*

The number to check for evenness.

odd

Determines whether an integer is odd.

```
calc.odd( int ) -> bool
```

```
#calc.odd(4) \
#calc.odd(5) \
#range(10).filter(calc.odd)
```

```
false
```

```
true
(1, 3, 5, 7, 9)
```

value `int` *Required* *Positional*

The number to check for oddness.

rem

Calculates the remainder of two numbers.

The value `calc.rem(x, y)` always has the same sign as x, and is smaller in magnitude than y.

```
calc.rem(
  int float,
  int float,
) -> int float
```

```
#calc.rem(7, 3) \
#calc.rem(7, -3) \
#calc.rem(-7, 3) \
#calc.rem(-7, -3) \
#calc.rem(1.75, 0.5)
```

```
1
1
−1
−1
0.25
```

dividend `int` or `float` *Required* *Positional*

The dividend of the remainder.

divisor `int` or `float` *Required* *Positional*

The divisor of the remainder.

div-euclid

Performs euclidean division of two numbers.

The result of this computation is that of a division rounded to the integer `n` such that the dividend is greater than or equal to `n` times the divisor.

```
calc.div-euclid(
  int float,
  int float,
) -> int float
```

```
#calc.div-euclid(7, 3) \
#calc.div-euclid(7, -3) \
#calc.div-euclid(-7, 3) \
#calc.div-euclid(-7, -3) \
#calc.div-euclid(1.75, 0.5)
```

```
2
−2
−3
3
3
```

`dividend` `int` or `float` *Required* *Positional*

The dividend of the division.

`divisor` `int` or `float` *Required* *Positional*

The divisor of the division.

`rem-euclid`

This calculates the least nonnegative remainder of a division.

Warning: Due to a floating point round-off error, the remainder may equal the absolute value of the divisor if the dividend is much smaller in magnitude than the divisor and the dividend is negative. This only applies for floating point inputs.

```
calc.rem-euclid(
  int float,
  int float,
) -> int float
```

```
#calc.rem-euclid(7, 3) \
#calc.rem-euclid(7, -3) \
#calc.rem-euclid(-7, 3) \
#calc.rem-euclid(-7, -3) \
#calc.rem(1.75, 0.5)
```

```
1
1
2
2
0.25
```

dividend int or float *Required* *Positional*

The dividend of the remainder.

divisor int or float *Required* *Positional*

The divisor of the remainder.

quo

Calculates the quotient (floored division) of two numbers.

```
calc.quo(
  int float,
  int float,
) -> int
```

```
$ "quo"(a, b) &= floor(a/b) \
  "quo"(14, 5) &= #calc.quo(14, 5) \
  "quo"(3.46, 0.5) &= #calc.quo(3.46, 0.5) $
```

$$\text{quo}(a, b) = \left\lfloor \frac{a}{b} \right\rfloor$$

$$\text{quo}(14, 5) = 2$$
$$\text{quo}(3.46, 0.5) = 6$$

`dividend` `int` or `float` *Required* *Positional*

The dividend of the quotient.

`divisor` `int` or `float` *Required* *Positional*

The divisor of the quotient.

Bytes
Previous page

Content
Next page