Mathematics

Mathematical Operators

`Base.:-` — Method

```
-(x)
```

Unary minus operator.

Examples

```
julia> -1
-1

julia> -(2)
-2

julia> -[1 2; 3 4]
2×2 Array{Int64,2}:
 -1  -2
 -3  -4
```

`Base.:+` — Function

```
+(x, y...)
```

Addition operator. `x+y+z+...` calls this function with all arguments, i.e. `+(x, y, z, ...)`.

Examples

```
julia> 1 + 20 + 4
25
```

```
julia> +(1, 20, 4)
25
```

```
dt::Date + t::Time -> DateTime
```

The addition of a `Date` with a `Time` produces a `DateTime`. The hour, minute, second, and millisecond parts of the `Time` are used along with the year, month, and day of the `Date` to create the new `DateTime`. Non-zero microseconds or nanoseconds in the `Time` type will result in an `InexactError` being thrown.

`Base.:-` — Method

```
-(x, y)
```

Subtraction operator.

Examples

```
julia> 2 - 3
-1

julia> -(2, 4.5)
-2.5
```

`Base.:*` — Method

```
*(x, y...)
```

Multiplication operator. `x*y*z*...` calls this function with all arguments, i.e. `*(x, y, z, ...)`.

Examples

```
julia> 2 * 7 * 8
112

julia> *(2, 7, 8)
```

```
112
```

`Base.:/` — Function

```
/(x, y)
```

Right division operator: multiplication of `x` by the inverse of `y` on the right. Gives floating-point results for integer arguments.

Examples

```
julia> 1/2
0.5

julia> 4/2
2.0

julia> 4.5/2
2.25
```

`Base.:\` — Method

```
\(x, y)
```

Left division operator: multiplication of `y` by the inverse of `x` on the left. Gives floating-point results for integer arguments.

Examples

```
julia> 3 \ 6
2.0

julia> inv(3) * 6
2.0

julia> A = [4 3; 2 1]; x = [5, 6];

julia> A \ x
```

```
2-element Array{Float64,1}:
  6.5
 -7.0

julia> inv(A) * x
2-element Array{Float64,1}:
  6.5
 -7.0
```

Base.:^ — Method

```
^(x, y)
```

Exponentiation operator. If `x` is a matrix, computes matrix exponentiation.

If `y` is an `Int` literal (e.g. `2` in `x^2` or `-3` in `x^-3`), the Julia code `x^y` is transformed by the compiler to `Base.literal_pow(^, x, Val(y))`, to enable compile-time specialization on the value of the exponent. (As a default fallback we have `Base.literal_pow(^, x, Val(y)) = ^(x,y)`, where usually `^ == Base.^` unless `^` has been defined in the calling namespace.)

```
julia> 3^5
243

julia> A = [1 2; 3 4]
2×2 Array{Int64,2}:
 1  2
 3  4

julia> A^3
2×2 Array{Int64,2}:
 37   54
 81  118
```

Base.fma — Function

```
fma(x, y, z)
```

Computes `x*y+z` without rounding the intermediate result `x*y`. On some systems this is

significantly more expensive than `x*y+z`. `fma` is used to improve accuracy in certain algorithms. See `muladd`.

`Base.muladd` — Function

```
muladd(x, y, z)
```

Combined multiply-add: computes `x*y+z`, but allowing the add and multiply to be merged with each other or with surrounding operations for performance. For example, this may be implemented as an `fma` if the hardware supports it efficiently. The result can be different on different machines and can also be different on the same machine due to constant propagation or other optimizations. See `fma`.

Examples

```
julia> muladd(3, 2, 1)
7

julia> 3 * 2 + 1
7
```

`Base.inv` — Method

```
inv(x)
```

Return the multiplicative inverse of `x`, such that `x*inv(x)` or `inv(x)*x` yields `one(x)` (the multiplicative identity) up to roundoff errors.

If `x` is a number, this is essentially the same as `one(x)/x`, but for some types `inv(x)` may be slightly more efficient.

Examples

```
julia> inv(2)
0.5

julia> inv(1 + 2im)
0.2 - 0.4im
```

```
julia> inv(1 + 2im) * (1 + 2im)
1.0 + 0.0im

julia> inv(2//3)
3//2
```

Julia 1.2

`inv(::Missing)` requires at least Julia 1.2.

`Base.div` — Function

```
div(x, y)
÷(x, y)
```

The quotient from Euclidean division. Computes `x/y`, truncated to an integer.

Examples

```
julia> 9 ÷ 4
2

julia> -5 ÷ 3
-1

julia> 5.0 ÷ 2
2.0
```

`Base.fld` — Function

```
fld(x, y)
```

Largest integer less than or equal to `x/y`. Equivalent to `div(x, y, RoundDown)`.

See also: `div`

Examples

```
julia> fld(7.3,5.5)
1.0
```

Base.cld — Function

```
cld(x, y)
```

Smallest integer larger than or equal to `x/y`. Equivalent to `div(x, y, RoundUp)`.

See also: `div`

Examples

```
julia> cld(5.5,2.2)
3.0
```

Base.mod — Function

```
mod(x::Integer, r::AbstractUnitRange)
```

Find `y` in the range `r` such that $x \equiv y (mod n)$, where `n = length(r)`, i.e. `y = mod(x - first(r), n) + first(r)`.

See also: `mod1`.

Examples

```
julia> mod(0, Base.OneTo(3))
3

julia> mod(3, 0:2)
0
```

> **Julia 1.3**

This method requires at least Julia 1.3.

```
mod(x, y)
rem(x, y, RoundDown)
```

The reduction of `x` modulo `y`, or equivalently, the remainder of `x` after floored division by `y`, i.e. `x - y*fld(x,y)` if computed without intermediate rounding.

The result will have the same sign as `y`, and magnitude less than `abs(y)` (with some exceptions, see note below).

Note

When used with floating point values, the exact result may not be representable by the type, and so rounding error may occur. In particular, if the exact result is very close to `y`, then it may be rounded to `y`.

```
julia> mod(8, 3)
2

julia> mod(9, 3)
0

julia> mod(8.9, 3)
2.9000000000000004

julia> mod(eps(), 3)
2.220446049250313e-16

julia> mod(-eps(), 3)
3.0
```

```
rem(x::Integer, T::Type{<:Integer}) -> T
mod(x::Integer, T::Type{<:Integer}) -> T
%(x::Integer, T::Type{<:Integer}) -> T
```

Find `y::T` such that `x ≡ y (mod n)`, where n is the number of integers representable in `T`, and `y` is

an integer in `[typemin(T),typemax(T)]`. If `T` can represent any integer (e.g. `T == BigInt`), then this operation corresponds to a conversion to `T`.

Examples

```
julia> 129 % Int8
-127
```

`Base.rem` — Function

```
rem(x, y)
%(x, y)
```

Remainder from Euclidean division, returning a value of the same sign as `x`, and smaller in magnitude than `y`. This value is always exact.

Examples

```
julia> x = 15; y = 4;

julia> x % y
3

julia> x == div(x, y) * y + rem(x, y)
true
```

`Base.Math.rem2pi` — Function

```
rem2pi(x, r::RoundingMode)
```

Compute the remainder of `x` after integer division by `2π`, with the quotient rounded according to the rounding mode `r`. In other words, the quantity

```
x - 2π*round(x/(2π),r)
```

without any intermediate rounding. This internally uses a high precision approximation of 2π, and so will give a more accurate result than `rem(x,2π,r)`

- if `r == RoundNearest`, then the result is in the interval $[-\pi, \pi]$. This will generally be the most accurate result. See also `RoundNearest`.
- if `r == RoundToZero`, then the result is in the interval $[0, 2\pi]$ if `x` is positive,. or $[-2\pi, 0]$ otherwise. See also `RoundToZero`.
- if `r == RoundDown`, then the result is in the interval $[0, 2\pi]$. See also `RoundDown`.
- if `r == RoundUp`, then the result is in the interval $[-2\pi, 0]$. See also `RoundUp`.

Examples

```
julia> rem2pi(7pi/4, RoundNearest)
-0.7853981633974485

julia> rem2pi(7pi/4, RoundDown)
5.497787143782138
```

`Base.Math.mod2pi` — Function

```
mod2pi(x)
```

Modulus after division by `2π`, returning in the range $[0, 2\pi)$.

This function computes a floating point representation of the modulus after division by numerically exact `2π`, and is therefore not exactly the same as `mod(x,2π)`, which would compute the modulus of `x` relative to division by the floating-point number `2π`.

> **Note**
>
> Depending on the format of the input value, the closest representable value to `2π` may be less than `2π`. For example, the expression `mod2pi(2π)` will not return `0`, because the intermediate value of `2*π` is a `Float64` and `2*Float64(π) < 2*big(π)`. See `rem2pi` for more refined control of this behavior.

Examples

```
julia> mod2pi(9*pi/4)
0.7853981633974481
```

`Base.divrem` — Function

```
divrem(x, y, r::RoundingMode=RoundToZero)
```

The quotient and remainder from Euclidean division. Equivalent to `(div(x,y,r), rem(x,y,r))`. Equivalently, with the default value of `r`, this call is equivalent to `(x÷y, x%y)`.

Examples

```
julia> divrem(3,7)
(0, 3)

julia> divrem(7,3)
(2, 1)
```

`Base.fldmod` — Function

```
fldmod(x, y)
```

The floored quotient and modulus after division. A convenience wrapper for `divrem(x, y, RoundDown)`. Equivalent to `(fld(x,y), mod(x,y))`.

`Base.fld1` — Function

```
fld1(x, y)
```

Flooring division, returning a value consistent with `mod1(x,y)`

See also: `mod1`, `fldmod1`.

Examples

```
julia> x = 15; y = 4;

julia> fld1(x, y)
4
```

```
julia> x == fld(x, y) * y + mod(x, y)
true

julia> x == (fld1(x, y) - 1) * y + mod1(x, y)
true
```

Base.mod1 — Function

```
mod1(x, y)
```

Modulus after flooring division, returning a value `r` such that `mod(r, y) == mod(x, y)` in the range $(0, y]$ for positive `y` and in the range $[y, 0)$ for negative `y`.

See also: `fld1`, `fldmod1`.

Examples

```
julia> mod1(4, 2)
2

julia> mod1(4, 3)
1
```

Base.fldmod1 — Function

```
fldmod1(x, y)
```

Return `(fld1(x,y), mod1(x,y))`.

See also: `fld1`, `mod1`.

Base.// — Function

```
//(num, den)
```

Divide two integers or rational numbers, giving a `Rational` result.

Examples

```
julia> 3 // 5
3//5

julia> (3 // 5) // (2 // 1)
3//10
```

`Base.rationalize` — Function

```
rationalize([T<:Integer=Int,] x; tol::Real=eps(x))
```

Approximate floating point number `x` as a `Rational` number with components of the given integer type. The result will differ from `x` by no more than `tol`.

Examples

```
julia> rationalize(5.6)
28//5

julia> a = rationalize(BigInt, 10.3)
103//10

julia> typeof(numerator(a))
BigInt
```

`Base.numerator` — Function

```
numerator(x)
```

Numerator of the rational representation of `x`.

Examples

```
julia> numerator(2//3)
2

julia> numerator(4)
```

```
4
```

`Base.denominator` — Function

```
denominator(x)
```

Denominator of the rational representation of `x`.

Examples

```
julia> denominator(2//3)
3

julia> denominator(4)
1
```

`Base.:<<` — Function

```
<<(x, n)
```

Left bit shift operator, `x << n`. For `n >= 0`, the result is `x` shifted left by `n` bits, filling with `0`s. This is equivalent to `x * 2^n`. For `n < 0`, this is equivalent to `x >> -n`.

Examples

```
julia> Int8(3) << 2
12

julia> bitstring(Int8(3))
"00000011"

julia> bitstring(Int8(12))
"00001100"
```

See also `>>`, `>>>`.

```
<<(B::BitVector, n) -> BitVector
```

Left bit shift operator, `B << n`. For `n >= 0`, the result is `B` with elements shifted `n` positions backwards, filling with `false` values. If `n < 0`, elements are shifted forwards. Equivalent to `B >> -n`.

Examples

```
julia> B = BitVector([true, false, true, false, false])
5-element BitArray{1}:
 1
 0
 1
 0
 0

julia> B << 1
5-element BitArray{1}:
 0
 1
 0
 0
 0

julia> B << -1
5-element BitArray{1}:
 0
 1
 0
 1
 0
```

`Base.:>>` — Function

```
>>(x, n)
```

Right bit shift operator, `x >> n`. For `n >= 0`, the result is `x` shifted right by `n` bits, where `n >= 0`, filling with `0`s if `x >= 0`, `1`s if `x < 0`, preserving the sign of `x`. This is equivalent to `fld(x, 2^n)`. For `n < 0`, this is equivalent to `x << -n`.

Examples

```
julia> Int8(13) >> 2
3

julia> bitstring(Int8(13))
"00001101"

julia> bitstring(Int8(3))
"00000011"

julia> Int8(-14) >> 2
-4

julia> bitstring(Int8(-14))
"11110010"

julia> bitstring(Int8(-4))
"11111100"
```

See also `>>>`, `<<`.

```
>>(B::BitVector, n) -> BitVector
```

Right bit shift operator, `B >> n`. For `n >= 0`, the result is `B` with elements shifted `n` positions forward, filling with `false` values. If `n < 0`, elements are shifted backwards. Equivalent to `B << -n`.

Examples

```
julia> B = BitVector([true, false, true, false, false])
5-element BitArray{1}:
 1
 0
 1
 0
 0

julia> B >> 1
5-element BitArray{1}:
 0
 1
```

```
 0
 1
 0

julia> B >> -1
5-element BitArray{1}:
 0
 1
 0
 0
 0
```

`Base.:>>>` — Function

```
>>>(x, n)
```

Unsigned right bit shift operator, `x >>> n`. For `n >= 0`, the result is `x` shifted right by `n` bits, where `n >= 0`, filling with `0`s. For `n < 0`, this is equivalent to `x << -n`.

For `Unsigned` integer types, this is equivalent to `>>`. For `Signed` integer types, this is equivalent to `signed(unsigned(x) >> n)`.

Examples

```
julia> Int8(-14) >>> 2
60

julia> bitstring(Int8(-14))
"11110010"

julia> bitstring(Int8(60))
"00111100"
```

`BigInt`s are treated as if having infinite size, so no filling is required and this is equivalent to `>>`.

See also `>>`, `<<`.

```
>>>(B::BitVector, n) -> BitVector
```

Unsigned right bitshift operator, `B >>> n`. Equivalent to `B >> n`. See `>>` for details and examples.

`Base.bitrotate` — Function

```
bitrotate(x::Base.BitInteger, k::Integer)
```

`bitrotate(x, k)` implements bitwise rotation. It returns the value of `x` with its bits rotated left `k` times. A negative value of `k` will rotate to the right instead.

> **Julia 1.5**
>
> This function requires Julia 1.5 or later.

```
julia> bitrotate(UInt8(114), 2)
0xc9

julia> bitstring(bitrotate(0b01110010, 2))
"11001001"

julia> bitstring(bitrotate(0b01110010, -2))
"10011100"

julia> bitstring(bitrotate(0b01110010, 8))
"01110010"
```

`Base.::` — Function

```
(:)(I::CartesianIndex, J::CartesianIndex)
```

Construct `CartesianIndices` from two `CartesianIndex`.

> **Julia 1.1**

This method requires at least Julia 1.1.

Examples

```
julia> I = CartesianIndex(2,1);

julia> J = CartesianIndex(3,3);

julia> I:J
2×3 CartesianIndices{2,Tuple{UnitRange{Int64},UnitRange{Int64}}}:
 CartesianIndex(2, 1)  CartesianIndex(2, 2)  CartesianIndex(2, 3)
 CartesianIndex(3, 1)  CartesianIndex(3, 2)  CartesianIndex(3, 3)
```

```
(:)(start, [step], stop)
```

Range operator. `a:b` constructs a range from `a` to `b` with a step size of 1 (a `UnitRange`), and `a:s:b` is similar but uses a step size of `s` (a `StepRange`).

`:` is also used in indexing to select whole dimensions and for `Symbol` literals, as in e.g. `:hello`.

`Base.range` — Function

```
range(start[, stop]; length, stop, step=1)
```

Given a starting value, construct a range either by length or from `start` to `stop`, optionally with a given step (defaults to 1, a `UnitRange`). One of `length` or `stop` is required. If `length`, `stop`, and `step` are all specified, they must agree.

If `length` and `stop` are provided and `step` is not, the step size will be computed automatically such that there are `length` linearly spaced elements in the range.

If `step` and `stop` are provided and `length` is not, the overall range length will be computed automatically such that the elements are `step` spaced.

Special care is taken to ensure intermediate values are computed rationally. To avoid this induced overhead, see the `LinRange` constructor.

`stop` may be specified as either a positional or keyword argument.

Julia 1.1

`stop` as a positional argument requires at least Julia 1.1.

Examples

```
julia> range(1, length=100)
1:100

julia> range(1, stop=100)
1:100

julia> range(1, step=5, length=100)
1:5:496

julia> range(1, step=5, stop=100)
1:5:96

julia> range(1, 10, length=101)
1.0:0.09:10.0

julia> range(1, 100, step=5)
1:5:96
```

`Base.OneTo` — Type

```
Base.OneTo(n)
```

Define an `AbstractUnitRange` that behaves like `1:n`, with the added distinction that the lower limit is guaranteed (by the type system) to be 1.

`Base.StepRangeLen` — Type

```
StepRangeLen{T,R,S}(ref::R, step::S, len, [offset=1]) where {T,R,S}
StepRangeLen(       ref::R, step::S, len, [offset=1]) where {  R,S}
```

A range `r` where `r[i]` produces values of type `T` (in the second form, `T` is deduced automatically),

parameterized by a `ref`erence value, a `step`, and the `len`gth. By default `ref` is the starting value `r[1]`, but alternatively you can supply it as the value of `r[offset]` for some other index `1 <= offset <= len`. In conjunction with `TwicePrecision` this can be used to implement ranges that are free of roundoff error.

`Base.:==` — Function

```
==(x, y)
```

Generic equality operator. Falls back to `===`. Should be implemented for all types with a notion of equality, based on the abstract value that an instance represents. For example, all numeric types are compared by numeric value, ignoring type. Strings are compared as sequences of characters, ignoring encoding. For collections, `==` is generally called recursively on all contents, though other properties (like the shape for arrays) may also be taken into account.

This operator follows IEEE semantics for floating-point numbers: `0.0 == -0.0` and `NaN != NaN`.

The result is of type `Bool`, except when one of the operands is `missing`, in which case `missing` is returned (three-valued logic). For collections, `missing` is returned if at least one of the operands contains a `missing` value and all non-missing values are equal. Use `isequal` or `===` to always get a `Bool` result.

Implementation

New numeric types should implement this function for two arguments of the new type, and handle comparison to other types via promotion rules where possible.

`isequal` falls back to `==`, so new methods of `==` will be used by the `Dict` type to compare keys. If your type will be used as a dictionary key, it should therefore also implement `hash`.

```
==(x)
```

Create a function that compares its argument to `x` using `==`, i.e. a function equivalent to `y -> y == x`.

The returned function is of type `Base.Fix2{typeof(==)}`, which can be used to implement specialized methods.

```
==(a::AbstractString, b::AbstractString) -> Bool
```

Test whether two strings are equal character by character (technically, Unicode code point by code point).

Examples

```
julia> "abc" == "abc"
true

julia> "abc" == "αβγ"
false
```

`Base.:!=` — Function

```
!=(x, y)
≠(x,y)
```

Not-equals comparison operator. Always gives the opposite answer as `==`.

Implementation

New types should generally not implement this, and rely on the fallback definition `!=(x,y) = !(x==y)` instead.

Examples

```
julia> 3 != 2
true

julia> "foo" ≠ "foo"
false
```

```
!=(x)
```

Create a function that compares its argument to `x` using `!=`, i.e. a function equivalent to `y -> y != x`. The returned function is of type `Base.Fix2{typeof(!=)}`, which can be used to implement

specialized methods.

> **Julia 1.2**
>
> This functionality requires at least Julia 1.2.

`Base.:!==` — Function

```
!==(x, y)
≠(x,y)
```

Always gives the opposite answer as `===`.

Examples

```
julia> a = [1, 2]; b = [1, 2];

julia> a ≠ b
true

julia> a ≠ a
false
```

`Base.:<` — Function

```
<(x, y)
```

Less-than comparison operator. Falls back to `isless`. Because of the behavior of floating-point NaN values, this operator implements a partial order.

Implementation

New numeric types with a canonical partial order should implement this function for two arguments of the new type. Types with a canonical total order should implement `isless` instead.
(x < y) | (x == y)

Examples

```
julia> 'a' < 'b'
true

julia> "abc" < "abd"
true

julia> 5 < 3
false
```

```
<(x)
```

Create a function that compares its argument to `x` using `<`, i.e. a function equivalent to `y -> y < x`. The returned function is of type `Base.Fix2{typeof(<)}`, which can be used to implement specialized methods.

> **Julia 1.2**
>
> This functionality requires at least Julia 1.2.

`Base.:<=` — Function

```
<=(x, y)
≤(x,y)
```

Less-than-or-equals comparison operator. Falls back to `(x < y) | (x == y)`.

Examples

```
julia> 'a' <= 'b'
true

julia> 7 ≤ 7 ≤ 9
true

julia> "abc" ≤ "abc"
true

julia> 5 <= 3
```

```
false
```

```
<=(x)
```

Create a function that compares its argument to `x` using `<=`, i.e. a function equivalent to `y -> y <= x`. The returned function is of type `Base.Fix2{typeof(<=)}`, which can be used to implement specialized methods.

> **Julia 1.2**
>
> This functionality requires at least Julia 1.2.

`Base.:>` — Function

```
>(x, y)
```

Greater-than comparison operator. Falls back to `y < x`.

Implementation

Generally, new types should implement `<` instead of this function, and rely on the fallback definition `>(x, y) = y < x`.

Examples

```
julia> 'a' > 'b'
false

julia> 7 > 3 > 1
true

julia> "abc" > "abd"
false

julia> 5 > 3
true
```

```
>(x)
```

Create a function that compares its argument to `x` using `>`, i.e. a function equivalent to `y -> y > x`. The returned function is of type `Base.Fix2{typeof(>)}`, which can be used to implement specialized methods.

> **Julia 1.2**
>
> This functionality requires at least Julia 1.2.

`Base.:>=` — Function

```
>=(x, y)
≥(x,y)
```

Greater-than-or-equals comparison operator. Falls back to `y <= x`.

Examples

```
julia> 'a' >= 'b'
false

julia> 7 ≥ 7 ≥ 3
true

julia> "abc" ≥ "abc"
true

julia> 5 >= 3
true
```

```
>=(x)
```

Create a function that compares its argument to `x` using `>=`, i.e. a function equivalent to `y -> y >= x`. The returned function is of type `Base.Fix2{typeof(>=)}`, which can be used to implement specialized methods.

> **Julia 1.2**
>
> This functionality requires at least Julia 1.2.

`Base.cmp` — Function

```
cmp(x,y)
```

Return -1, 0, or 1 depending on whether `x` is less than, equal to, or greater than `y`, respectively. Uses the total order implemented by `isless`.

Examples

```
julia> cmp(1, 2)
-1

julia> cmp(2, 1)
1

julia> cmp(2+im, 3-im)
ERROR: MethodError: no method matching isless(::Complex{Int64}, ::Complex{Int64
[...]
```

```
cmp(<, x, y)
```

Return -1, 0, or 1 depending on whether `x` is less than, equal to, or greater than `y`, respectively. The first argument specifies a less-than comparison function to use.

```
cmp(a::AbstractString, b::AbstractString) -> Int
```

Compare two strings. Return `0` if both strings have the same length and the character at each index is the same in both strings. Return `-1` if `a` is a prefix of `b`, or if `a` comes before `b` in alphabetical order. Return `1` if `b` is a prefix of `a`, or if `b` comes before `a` in alphabetical order (technically, lexicographical order by Unicode code points).

Examples

```
julia> cmp("abc", "abc")
0

julia> cmp("ab", "abc")
-1

julia> cmp("abc", "ab")
1

julia> cmp("ab", "ac")
-1

julia> cmp("ac", "ab")
1

julia> cmp("α", "a")
1

julia> cmp("b", "β")
-1
```

`Base.:~` — Function

```
~(x)
```

Bitwise not.

Examples

```
julia> ~4
-5

julia> ~10
-11

julia> ~true
false
```

`Base.:&` — Function

```
x & y
```

Bitwise and. Implements three-valued logic, returning `missing` if one operand is `missing` and the other is `true`. Add parentheses for function application form: `(&)(x, y)`.

Examples

```
julia> 4 & 10
0

julia> 4 & 12
4

julia> true & missing
missing

julia> false & missing
false
```

`Base.:|` — Function

```
x | y
```

Bitwise or. Implements three-valued logic, returning `missing` if one operand is `missing` and the other is `false`.

Examples

```
julia> 4 | 10
14

julia> 4 | 1
5

julia> true | missing
true

julia> false | missing
missing
```

`Base.xor` — Function

```
xor(x, y)
⊻(x, y)
```

Bitwise exclusive or of `x` and `y`. Implements three-valued logic, returning `missing` if one of the arguments is `missing`.

The infix operation `a ⊻ b` is a synonym for `xor(a,b)`, and `⊻` can be typed by tab-completing `\xor` or `\veebar` in the Julia REPL.

Examples

```
julia> xor(true, false)
true

julia> xor(true, true)
false

julia> xor(true, missing)
missing
```

```
julia> false ⩔ false
false

julia> [true; true; false] .⩔ [true; false; false]
3-element BitArray{1}:
 0
 1
 0
```

`Base.:!` — Function

```
!(x)
```

Boolean not. Implements three-valued logic, returning `missing` if `x` is `missing`.

Examples

```
julia> !true
false

julia> !false
true

julia> !missing
missing

julia> .![true false true]
1×3 BitArray{2}:
 0  1  0
```

```
!f::Function
```

Predicate function negation: when the argument of `!` is a function, it returns a function which computes the boolean negation of `f`.

Examples

```
julia> str = "∀ ε > 0, ∃ δ > 0: |x-y| < δ ⇒ |f(x)-f(y)| < ε"
```

```
"∀ ε > 0, ∃ δ > 0: |x-y| < δ ⇒ |f(x)-f(y)| < ε"

julia> filter(isletter, str)
"εδxyδfxfyε"

julia> filter(!isletter, str)
"∀  > 0, ∃  > 0: |-| <  ⇒ |()-()| < "
```

`&&` — Keyword

```
x && y
```

Short-circuiting boolean AND.

`||` — Keyword

```
x || y
```

Short-circuiting boolean OR.

Mathematical Functions

`Base.isapprox` — Function

```
isapprox(x, y; rtol::Real=atol>0 ? 0 : √eps, atol::Real=0, nans::Bool=false, no
```

Inexact equality comparison: `true` if `norm(x-y) <= max(atol, rtol*max(norm(x), norm(y)))`. The default `atol` is zero and the default `rtol` depends on the types of `x` and `y`. The keyword argument `nans` determines whether or not NaN values are considered equal (defaults to false).

For real or complex floating-point values, if an `atol > 0` is not specified, `rtol` defaults to the square root of `eps` of the type of `x` or `y`, whichever is bigger (least precise). This corresponds to requiring equality of about half of the significand digits. Otherwise, e.g. for integer arguments or if an `atol > 0` is supplied, `rtol` defaults to zero.

`x` and `y` may also be arrays of numbers, in which case `norm` defaults to the usual `norm` function in LinearAlgebra, but may be changed by passing a `norm::Function` keyword argument. (For numbers, `norm` is the same thing as `abs`.) When `x` and `y` are arrays, if `norm(x-y)` is not finite (i.e. `±Inf` or `NaN`), the comparison falls back to checking whether all elements of `x` and `y` are approximately equal component-wise.

The binary operator ≈ is equivalent to `isapprox` with the default arguments, and `x ≉ y` is equivalent to `!isapprox(x,y)`.

Note that `x ≈ 0` (i.e., comparing to zero with the default tolerances) is equivalent to `x == 0` since the default `atol` is `0`. In such cases, you should either supply an appropriate `atol` (or use `norm(x) ≤ atol`) or rearrange your code (e.g. use `x ≈ y` rather than `x - y ≈ 0`). It is not possible to pick a nonzero `atol` automatically because it depends on the overall scaling (the "units") of your problem: for example, in `x - y ≈ 0`, `atol=1e-9` is an absurdly small tolerance if `x` is the radius of the Earth in meters, but an absurdly large tolerance if `x` is the radius of a Hydrogen atom in meters.

Examples

```
julia> 0.1 ≈ (0.1 - 1e-10)
true

julia> isapprox(10, 11; atol = 2)
true

julia> isapprox([10.0^9, 1.0], [10.0^9, 2.0])
true

julia> 1e-10 ≈ 0
false

julia> isapprox(1e-10, 0, atol=1e-8)
true
```

```
isapprox(x; kwargs...) / ≈(x; kwargs...)
```

Create a function that compares its argument to `x` using ≈, i.e. a function equivalent to `y -> y ≈ x`.

The keyword arguments supported here are the same as those in the 2-argument `isapprox`.

`Base.sin` — Method

```
sin(x)
```

Compute sine of `x`, where `x` is in radians.

`Base.cos` — Method

```
cos(x)
```

Compute cosine of `x`, where `x` is in radians.

`Base.Math.sincos` — Method

```
sincos(x)
```

Simultaneously compute the sine and cosine of `x`, where the `x` is in radians.

`Base.tan` — Method

```
tan(x)
```

Compute tangent of `x`, where `x` is in radians.

`Base.Math.sind` — Function

```
sind(x)
```

Compute sine of `x`, where `x` is in degrees.

`Base.Math.cosd` — Function

```
cosd(x)
```

Compute cosine of `x`, where `x` is in degrees.

`Base.Math.tand` — Function

```
tand(x)
```

Compute tangent of `x`, where `x` is in degrees.

`Base.Math.sinpi` — Function

```
sinpi(x)
```

Compute $\sin(\pi x)$ more accurately than `sin(pi*x)`, especially for large `x`.

`Base.Math.cospi` — Function

```
cospi(x)
```

Compute $\cos(\pi x)$ more accurately than `cos(pi*x)`, especially for large `x`.

`Base.sinh` — Method

```
sinh(x)
```

Compute hyperbolic sine of `x`.

`Base.cosh` — Method

```
cosh(x)
```

Compute hyperbolic cosine of `x`.

`Base.tanh` — Method

```
tanh(x)
```

Compute hyperbolic tangent of `x`.

`Base.asin` — Method

```
asin(x)
```

Compute the inverse sine of `x`, where the output is in radians.

`Base.acos` — Method

```
acos(x)
```

Compute the inverse cosine of `x`, where the output is in radians

`Base.atan` — Method

```
atan(y)
atan(y, x)
```

Compute the inverse tangent of `y` or `y/x`, respectively.

For one argument, this is the angle in radians between the positive *x*-axis and the point (1, *y*), returning a value in the interval $[-\pi/2, \pi/2]$.

For two arguments, this is the angle in radians between the positive *x*-axis and the point (*x*, *y*),

returning a value in the interval $[-\pi, \pi]$. This corresponds to a standard `atan2` function.

`Base.Math.asind` — Function

```
asind(x)
```

Compute the inverse sine of `x`, where the output is in degrees.

`Base.Math.acosd` — Function

```
acosd(x)
```

Compute the inverse cosine of `x`, where the output is in degrees.

`Base.Math.atand` — Function

```
atand(y)
atand(y,x)
```

Compute the inverse tangent of `y` or `y/x`, respectively, where the output is in degrees.

`Base.Math.sec` — Method

```
sec(x)
```

Compute the secant of `x`, where `x` is in radians.

`Base.Math.csc` — Method

```
csc(x)
```

Compute the cosecant of `x`, where `x` is in radians.

`Base.Math.cot` — Method

```
cot(x)
```

Compute the cotangent of `x`, where `x` is in radians.

`Base.Math.secd` — Function

```
secd(x)
```

Compute the secant of `x`, where `x` is in degrees.

`Base.Math.cscd` — Function

```
cscd(x)
```

Compute the cosecant of `x`, where `x` is in degrees.

`Base.Math.cotd` — Function

```
cotd(x)
```

Compute the cotangent of `x`, where `x` is in degrees.

`Base.Math.asec` — Method

```
asec(x)
```

Compute the inverse secant of `x`, where the output is in radians.

`Base.Math.acsc` — Method

```
acsc(x)
```

Compute the inverse cosecant of `x`, where the output is in radians.

`Base.Math.acot` — Method

```
acot(x)
```

Compute the inverse cotangent of `x`, where the output is in radians.

`Base.Math.asecd` — Function

```
asecd(x)
```

Compute the inverse secant of `x`, where the output is in degrees.

`Base.Math.acscd` — Function

```
acscd(x)
```

Compute the inverse cosecant of `x`, where the output is in degrees.

`Base.Math.acotd` — Function

```
acotd(x)
```

Compute the inverse cotangent of `x`, where the output is in degrees.

`Base.Math.sech` — Method

```
sech(x)
```

Compute the hyperbolic secant of `x`.

`Base.Math.csch` — Method

```
csch(x)
```

Compute the hyperbolic cosecant of `x`.

`Base.Math.coth` — Method

```
coth(x)
```

Compute the hyperbolic cotangent of `x`.

`Base.asinh` — Method

```
asinh(x)
```

Compute the inverse hyperbolic sine of `x`.

`Base.acosh` — Method

```
acosh(x)
```

Compute the inverse hyperbolic cosine of `x`.

`Base.atanh` — Method

```
atanh(x)
```

Compute the inverse hyperbolic tangent of `x`.

`Base.Math.asech` — Method

```
asech(x)
```

Compute the inverse hyperbolic secant of `x`.

`Base.Math.acsch` — Method

```
acsch(x)
```

Compute the inverse hyperbolic cosecant of `x`.

`Base.Math.acoth` — Method

```
acoth(x)
```

Compute the inverse hyperbolic cotangent of `x`.

`Base.Math.sinc` — Function

```
sinc(x)
```

Compute $\sin(\pi x)/(\pi x)$ if $x \neq 0$, and 1 if $x = 0$.

Base.Math.cosc — Function

```
cosc(x)
```

Compute $\cos(\pi x)/x - \sin(\pi x)/(\pi x^2)$ if $x \neq 0$, and 0 if $x = 0$. This is the derivative of `sinc(x)`.

Base.Math.deg2rad — Function

```
deg2rad(x)
```

Convert `x` from degrees to radians.

Examples

```
julia> deg2rad(90)
1.5707963267948966
```

Base.Math.rad2deg — Function

```
rad2deg(x)
```

Convert `x` from radians to degrees.

Examples

```
julia> rad2deg(pi)
180.0
```

`Base.Math.hypot` — Function

```
hypot(x, y)
```

Compute the hypotenuse $\sqrt{|x|^2 + |y|^2}$ avoiding overflow and underflow.

This code is an implementation of the algorithm described in: An Improved Algorithm for `hypot(a,b)` by Carlos F. Borges The article is available online at ArXiv at the link https://arxiv.org/abs/1904.09481

Examples

```
julia> a = Int64(10)^10;

julia> hypot(a, a)
1.4142135623730951e10

julia> √(a^2 + a^2) # a^2 overflows
ERROR: DomainError with -2.914184810805068e18:
sqrt will only return a complex result if called with a complex argument. Try s
Stacktrace:
[...]

julia> hypot(3, 4im)
5.0
```

```
hypot(x...)
```

Compute the hypotenuse $\sqrt{\sum |x_i|^2}$ avoiding overflow and underflow.

Examples

```
julia> hypot(-5.7)
5.7

julia> hypot(3, 4im, 12.0)
13.0
```

`Base.log` — Method

```
log(x)
```

Compute the natural logarithm of `x`. Throws `DomainError` for negative `Real` arguments. Use complex negative arguments to obtain complex results.

Examples

```
julia> log(2)
0.6931471805599453

julia> log(-3)
ERROR: DomainError with -3.0:
log will only return a complex result if called with a complex argument. Try lo
Stacktrace:
 [1] throw_complex_domainerror(::Symbol, ::Float64) at ./math.jl:31
[...]
```

`Base.log` — Method

```
log(b,x)
```

Compute the base `b` logarithm of `x`. Throws `DomainError` for negative `Real` arguments.

Examples

```
julia> log(4,8)
1.5

julia> log(4,2)
0.5

julia> log(-2, 3)
ERROR: DomainError with -2.0:
log will only return a complex result if called with a complex argument. Try lo
Stacktrace:
 [1] throw_complex_domainerror(::Symbol, ::Float64) at ./math.jl:31
[...]
```

```
julia> log(2, -3)
ERROR: DomainError with -3.0:
log will only return a complex result if called with a complex argument. Try lo
Stacktrace:
 [1] throw_complex_domainerror(::Symbol, ::Float64) at ./math.jl:31
[...]
```

Note

If `b` is a power of 2 or 10, `log2` or `log10` should be used, as these will typically be faster and more accurate. For example,

```
julia> log(100,1000000)
2.9999999999999996

julia> log10(1000000)/2
3.0
```

`Base.log2` — Function

```
log2(x)
```

Compute the logarithm of `x` to base 2. Throws `DomainError` for negative `Real` arguments.

Examples

```
julia> log2(4)
2.0

julia> log2(10)
3.321928094887362

julia> log2(-2)
ERROR: DomainError with -2.0:
NaN result for non-NaN input.
Stacktrace:
 [1] nan_dom_err at ./math.jl:325 [inlined]
[...]
```

`Base.log10` — Function

```
log10(x)
```

Compute the logarithm of `x` to base 10. Throws `DomainError` for negative `Real` arguments.

Examples

```
julia> log10(100)
2.0

julia> log10(2)
0.3010299956639812

julia> log10(-2)
ERROR: DomainError with -2.0:
NaN result for non-NaN input.
Stacktrace:
 [1] nan_dom_err at ./math.jl:325 [inlined]
[...]
```

Base.log1p — Function

```
log1p(x)
```

Accurate natural logarithm of `1+x`. Throws `DomainError` for `Real` arguments less than -1.

Examples

```
julia> log1p(-0.5)
-0.6931471805599453

julia> log1p(0)
0.0

julia> log1p(-2)
ERROR: DomainError with -2.0:
log1p will only return a complex result if called with a complex argument. Try
Stacktrace:
 [1] throw_complex_domainerror(::Symbol, ::Float64) at ./math.jl:31
[...]
```

Base.Math.frexp — Function

```
frexp(val)
```

Return `(x, exp)` such that `x` has a magnitude in the interval $[1/2, 1)$ or 0, and `val` is equal to $x \times 2^{exp}$.

Base.exp — Method

```
exp(x)
```

Compute the natural base exponential of `x`, in other words e^x.

Examples

```
julia> exp(1.0)
```

```
2.718281828459045
```

Base.exp2 — Function

```
exp2(x)
```

Compute the base 2 exponential of `x`, in other words 2^x.

Examples

```
julia> exp2(5)
32.0
```

Base.exp10 — Function

```
exp10(x)
```

Compute the base 10 exponential of `x`, in other words 10^x.

Examples

```
julia> exp10(2)
100.0
```

```
exp10(x)
```

Compute 10^x.

Examples

```
julia> exp10(2)
100.0

julia> exp10(0.2)
1.5848931924611136
```

Base.Math.ldexp — Function

```
ldexp(x, n)
```

Compute $x \times 2^n$.

Examples

```
julia> ldexp(5., 2)
20.0
```

Base.Math.modf — Function

```
modf(x)
```

Return a tuple `(fpart, ipart)` of the fractional and integral parts of a number. Both parts have the same sign as the argument.

Examples

```
julia> modf(3.5)
(0.5, 3.0)

julia> modf(-3.5)
(-0.5, -3.0)
```

Base.expm1 — Function

```
expm1(x)
```

Accurately compute $e^x - 1$.

Base.round — Method

```
round([T,] x, [r::RoundingMode])
round(x, [r::RoundingMode]; digits::Integer=0, base = 10)
round(x, [r::RoundingMode]; sigdigits::Integer, base = 10)
```

Rounds the number `x`.

Without keyword arguments, `x` is rounded to an integer value, returning a value of type `T`, or of the same type of `x` if no `T` is provided. An `InexactError` will be thrown if the value is not representable by `T`, similar to `convert`.

If the `digits` keyword argument is provided, it rounds to the specified number of digits after the decimal place (or before if negative), in base `base`.

If the `sigdigits` keyword argument is provided, it rounds to the specified number of significant digits, in base `base`.

The `RoundingMode r` controls the direction of the rounding; the default is `RoundNearest`, which rounds to the nearest integer, with ties (fractional values of 0.5) being rounded to the nearest even integer. Note that `round` may give incorrect results if the global rounding mode is changed (see `rounding`).

Examples

```
julia> round(1.7)
2.0

julia> round(Int, 1.7)
2

julia> round(1.5)
2.0

julia> round(2.5)
2.0

julia> round(pi; digits=2)
3.14

julia> round(pi; digits=3, base=2)
3.125

julia> round(123.456; sigdigits=2)
120.0
```

```
julia> round(357.913; sigdigits=4, base=2)
352.0
```

Note

Rounding to specified digits in bases other than 2 can be inexact when operating on binary floating point numbers. For example, the `Float64` value represented by `1.15` is actually *less* than 1.15, yet will be rounded to 1.2.

Examples

```
julia> x = 1.15
1.15

julia> @sprintf "%.20f" x
"1.14999999999999991118"

julia> x < 115//100
true

julia> round(x, digits=1)
1.2
```

Extensions

To extend `round` to new numeric types, it is typically sufficient to define `Base.round(x::NewType, r::RoundingMode)`.

`Base.Rounding.RoundingMode` — Type

```
RoundingMode
```

A type used for controlling the rounding mode of floating point operations (via `rounding`/`setrounding` functions), or as optional arguments for rounding to the nearest integer (via the `round` function).

Currently supported rounding modes are:

- `RoundNearest` (default)
- `RoundNearestTiesAway`
- `RoundNearestTiesUp`
- `RoundToZero`
- `RoundFromZero` (`BigFloat` only)
- `RoundUp`
- `RoundDown`

`Base.Rounding.RoundNearest` — Constant

```
RoundNearest
```

The default rounding mode. Rounds to the nearest integer, with ties (fractional values of 0.5) being rounded to the nearest even integer.

`Base.Rounding.RoundNearestTiesAway` — Constant

```
RoundNearestTiesAway
```

Rounds to nearest integer, with ties rounded away from zero (C/C++ `round` behaviour).

`Base.Rounding.RoundNearestTiesUp` — Constant

```
RoundNearestTiesUp
```

Rounds to nearest integer, with ties rounded toward positive infinity (Java/JavaScript `round` behaviour).

`Base.Rounding.RoundToZero` — Constant

```
RoundToZero
```

`round` using this rounding mode is an alias for `trunc`.

`Base.Rounding.RoundFromZero` — Constant

```
RoundFromZero
```

Rounds away from zero. This rounding mode may only be used with `T == BigFloat` inputs to `round`.

Examples

```
julia> BigFloat("1.0000000000000001", 5, RoundFromZero)
1.06
```

`Base.Rounding.RoundUp` — Constant

```
RoundUp
```

`round` using this rounding mode is an alias for `ceil`.

`Base.Rounding.RoundDown` — Constant

```
RoundDown
```

`round` using this rounding mode is an alias for `floor`.

`Base.round` — Method

```
round(z::Complex[, RoundingModeReal, [RoundingModeImaginary]])
round(z::Complex[, RoundingModeReal, [RoundingModeImaginary]]; digits=, base=10
round(z::Complex[, RoundingModeReal, [RoundingModeImaginary]]; sigdigits=, base
```

Return the nearest integral value of the same type as the complex-valued `z` to `z`, breaking ties using the specified `RoundingMode`s. The first `RoundingMode` is used for rounding the real

components while the second is used for rounding the imaginary components.

Example

```
julia> round(3.14 + 4.5im)
3.0 + 4.0im
```

`Base.ceil` — Function

```
ceil([T,] x)
ceil(x; digits::Integer= [, base = 10])
ceil(x; sigdigits::Integer= [, base = 10])
```

`ceil(x)` returns the nearest integral value of the same type as `x` that is greater than or equal to `x`.

`ceil(T, x)` converts the result to type `T`, throwing an `InexactError` if the value is not representable.

`digits`, `sigdigits` and `base` work as for `round`.

`Base.floor` — Function

```
floor([T,] x)
floor(x; digits::Integer= [, base = 10])
floor(x; sigdigits::Integer= [, base = 10])
```

`floor(x)` returns the nearest integral value of the same type as `x` that is less than or equal to `x`.

`floor(T, x)` converts the result to type `T`, throwing an `InexactError` if the value is not representable.

`digits`, `sigdigits` and `base` work as for `round`.

`Base.trunc` — Function

```
trunc([T,] x)
```

```
trunc(x; digits::Integer= [, base = 10])
trunc(x; sigdigits::Integer= [, base = 10])
```

`trunc(x)` returns the nearest integral value of the same type as `x` whose absolute value is less than or equal to `x`.

`trunc(T, x)` converts the result to type `T`, throwing an `InexactError` if the value is not representable.

`digits`, `sigdigits` and `base` work as for `round`.

`Base.unsafe_trunc` — Function

```
unsafe_trunc(T, x)
```

Return the nearest integral value of type `T` whose absolute value is less than or equal to `x`. If the value is not representable by `T`, an arbitrary value will be returned.

`Base.min` — Function

```
min(x, y, ...)
```

Return the minimum of the arguments. See also the `minimum` function to take the minimum element from a collection.

Examples

```
julia> min(2, 5, 1)
1
```

`Base.max` — Function

```
max(x, y, ...)
```

Return the maximum of the arguments. See also the `maximum` function to take the maximum element from a collection.

Examples

```
julia> max(2, 5, 1)
5
```

Base.minmax — Function

```
minmax(x, y)
```

Return `(min(x,y), max(x,y))`. See also: `extrema` that returns `(minimum(x), maximum(x))`.

Examples

```
julia> minmax('c','b')
('b', 'c')
```

Base.Math.clamp — Function

```
clamp(x, lo, hi)
```

Return `x` if `lo <= x <= hi`. If `x > hi`, return `hi`. If `x < lo`, return `lo`. Arguments are promoted to a common type.

Examples

```
julia> clamp.([pi, 1.0, big(10.)], 2., 9.)
3-element Array{BigFloat,1}:
 3.14159265358979323846264338327950288419716939937510582097494459230781640628621
 2.0
 9.0

julia> clamp.([11,8,5],10,6) # an example where lo > hi
3-element Array{Int64,1}:
  6
  6
 10
```

```
clamp(x, T)::T
```

Clamp `x` between `typemin(T)` and `typemax(T)` and convert the result to type `T`.

Examples

```
julia> clamp(200, Int8)
127
julia> clamp(-200, Int8)
-128
```

`Base.Math.clamp!` — Function

```
clamp!(array::AbstractArray, lo, hi)
```

Restrict values in `array` to the specified range, in-place. See also `clamp`.

`Base.abs` — Function

```
abs(x)
```

The absolute value of `x`.

When `abs` is applied to signed integers, overflow may occur, resulting in the return of a negative value. This overflow occurs only when `abs` is applied to the minimum representable value of a signed integer. That is, when `x == typemin(typeof(x))`, `abs(x) == x < 0`, not `-x` as might be expected.

Examples

```
julia> abs(-3)
3

julia> abs(1 + im)
1.4142135623730951

julia> abs(typemin(Int64))
```

```
-9223372036854775808
```

Base.Checked.checked_abs — Function

```
Base.checked_abs(x)
```

Calculates `abs(x)`, checking for overflow errors where applicable. For example, standard two's complement signed integers (e.g. `Int`) cannot represent `abs(typemin(Int))`, thus leading to an overflow.

The overflow protection may impose a perceptible performance penalty.

Base.Checked.checked_neg — Function

```
Base.checked_neg(x)
```

Calculates `-x`, checking for overflow errors where applicable. For example, standard two's complement signed integers (e.g. `Int`) cannot represent `-typemin(Int)`, thus leading to an overflow.

The overflow protection may impose a perceptible performance penalty.

Base.Checked.checked_add — Function

```
Base.checked_add(x, y)
```

Calculates `x+y`, checking for overflow errors where applicable.

The overflow protection may impose a perceptible performance penalty.

Base.Checked.checked_sub — Function

```
Base.checked_sub(x, y)
```

Calculates `x-y`, checking for overflow errors where applicable.

The overflow protection may impose a perceptible performance penalty.

`Base.Checked.checked_mul` — Function

```
Base.checked_mul(x, y)
```

Calculates `x*y`, checking for overflow errors where applicable.

The overflow protection may impose a perceptible performance penalty.

`Base.Checked.checked_div` — Function

```
Base.checked_div(x, y)
```

Calculates `div(x,y)`, checking for overflow errors where applicable.

The overflow protection may impose a perceptible performance penalty.

`Base.Checked.checked_rem` — Function

```
Base.checked_rem(x, y)
```

Calculates `x%y`, checking for overflow errors where applicable.

The overflow protection may impose a perceptible performance penalty.

`Base.Checked.checked_fld` — Function

```
Base.checked_fld(x, y)
```

Calculates `fld(x,y)`, checking for overflow errors where applicable.

The overflow protection may impose a perceptible performance penalty.

`Base.Checked.checked_mod` — Function

```
Base.checked_mod(x, y)
```

Calculates `mod(x,y)`, checking for overflow errors where applicable.

The overflow protection may impose a perceptible performance penalty.

`Base.Checked.checked_cld` — Function

```
Base.checked_cld(x, y)
```

Calculates `cld(x,y)`, checking for overflow errors where applicable.

The overflow protection may impose a perceptible performance penalty.

`Base.Checked.add_with_overflow` — Function

```
Base.add_with_overflow(x, y) -> (r, f)
```

Calculates `r = x+y`, with the flag `f` indicating whether overflow has occurred.

`Base.Checked.sub_with_overflow` — Function

```
Base.sub_with_overflow(x, y) -> (r, f)
```

Calculates `r = x-y`, with the flag `f` indicating whether overflow has occurred.

Base.Checked.mul_with_overflow — Function

```
Base.mul_with_overflow(x, y) -> (r, f)
```

Calculates `r = x*y`, with the flag `f` indicating whether overflow has occurred.

Base.abs2 — Function

```
abs2(x)
```

Squared absolute value of `x`.

Examples

```
julia> abs2(-3)
9
```

Base.copysign — Function

```
copysign(x, y) -> z
```

Return `z` which has the magnitude of `x` and the same sign as `y`.

Examples

```
julia> copysign(1, -2)
-1

julia> copysign(-1, 2)
1
```

Base.sign — Function

```
sign(x)
```

Return zero if `x==0` and $x/|x|$ otherwise (i.e., ±1 for real `x`).

`Base.signbit` — Function

```
signbit(x)
```

Returns `true` if the value of the sign of `x` is negative, otherwise `false`.

Examples

```
julia> signbit(-4)
true

julia> signbit(5)
false

julia> signbit(5.5)
false

julia> signbit(-4.1)
true
```

`Base.flipsign` — Function

```
flipsign(x, y)
```

Return `x` with its sign flipped if `y` is negative. For example `abs(x) = flipsign(x,x)`.

Examples

```
julia> flipsign(5, 3)
5

julia> flipsign(5, -3)
-5
```

`Base.sqrt` — Method

```
sqrt(x)
```

Return $\sqrt{x}$. Throws `DomainError` for negative `Real` arguments. Use complex negative arguments instead. The prefix operator √ is equivalent to `sqrt`.

Examples

```
julia> sqrt(big(81))
9.0

julia> sqrt(big(-81))
ERROR: DomainError with -81.0:
NaN result for non-NaN input.
Stacktrace:
 [1] sqrt(::BigFloat) at ./mpfr.jl:501
[...]

julia> sqrt(big(complex(-81)))
0.0 + 9.0im
```

`Base.isqrt` — Function

```
isqrt(n::Integer)
```

Integer square root: the largest integer `m` such that `m*m <= n`.

```
julia> isqrt(5)
2
```

`Base.Math.cbrt` — Function

```
cbrt(x::Real)
```

Return the cube root of `x`, i.e. $x^{1/3}$. Negative values are accepted (returning the negative real root

when $x < 0$).

The prefix operator ∛ is equivalent to `cbrt`.

Examples

```
julia> cbrt(big(27))
3.0

julia> cbrt(big(-27))
-3.0
```

`Base.real` — Method

```
real(z)
```

Return the real part of the complex number `z`.

Examples

```
julia> real(1 + 3im)
1
```

`Base.imag` — Function

```
imag(z)
```

Return the imaginary part of the complex number `z`.

Examples

```
julia> imag(1 + 3im)
3
```

`Base.reim` — Function

```
reim(z)
```

Return both the real and imaginary parts of the complex number `z`.

Examples

```
julia> reim(1 + 3im)
(1, 3)
```

`Base.conj` — Function

```
conj(z)
```

Compute the complex conjugate of a complex number `z`.

Examples

```
julia> conj(1 + 3im)
1 - 3im
```

`Base.angle` — Function

```
angle(z)
```

Compute the phase angle in radians of a complex number `z`.

Examples

```
julia> rad2deg(angle(1 + im))
45.0

julia> rad2deg(angle(1 - im))
-45.0

julia> rad2deg(angle(-1 - im))
-135.0
```

`Base.cis` — Function

```
cis(z)
```

Return $\exp(iz)$.

Examples

```
julia> cis(π) ≈ -1
true
```

`Base.binomial` — Function

```
binomial(n::Integer, k::Integer)
```

The *binomial coefficient* $\binom{n}{k}$, being the coefficient of the kth term in the polynomial expansion of $(1+x)^n$.

If n is non-negative, then it is the number of ways to choose `k` out of `n` items:

$$\binom{n}{k} = \frac{n!}{k!(n-k)!}$$

where $n!$ is the `factorial` function.

If n is negative, then it is defined in terms of the identity

$$\binom{n}{k} = (-1)^k \binom{k-n-1}{k}$$

Examples

```
julia> binomial(5, 3)
10

julia> factorial(5) ÷ (factorial(5-3) * factorial(3))
10

julia> binomial(-5, 3)
-35
```

See also

- `factorial`

External links

- Binomial coefficient on Wikipedia.

`Base.factorial` — Function

```
factorial(n::Integer)
```

Factorial of `n`. If `n` is an `Integer`, the factorial is computed as an integer (promoted to at least 64 bits). Note that this may overflow if `n` is not small, but you can use `factorial(big(n))` to compute the result exactly in arbitrary precision.

Examples

```
julia> factorial(6)
720

julia> factorial(21)
ERROR: OverflowError: 21 is too large to look up in the table; consider using `
Stacktrace:
```

```
[...]

julia> factorial(big(21))
51090942171709440000
```

See also

- `binomial`

External links

- Factorial on Wikipedia.

`Base.gcd` — Function

```
gcd(x,y)
```

Greatest common (positive) divisor (or zero if `x` and `y` are both zero). The arguments may be integer and rational numbers.

> **Julia 1.4**
>
> Rational arguments require Julia 1.4 or later.

Examples

```
julia> gcd(6,9)
3

julia> gcd(6,-9)
3

julia> gcd(6,0)
6

julia> gcd(0,0)
0

julia> gcd(1//3,2//3)
1//3
```

```
julia> gcd(1//3,-2//3)
1//3

julia> gcd(1//3,2)
1//3
```

`Base.lcm` — Function

```
lcm(x,y)
```

Least common (non-negative) multiple. The arguments may be integer and rational numbers.

> **Julia 1.4**
>
> Rational arguments require Julia 1.4 or later.

Examples

```
julia> lcm(2,3)
6

julia> lcm(-2,3)
6

julia> lcm(0,3)
0

julia> lcm(0,0)
0

julia> lcm(1//3,2//3)
2//3

julia> lcm(1//3,-2//3)
2//3

julia> lcm(1//3,2)
2//1
```

`Base.gcdx` — Function

```
gcdx(x,y)
```

Computes the greatest common (positive) divisor of `x` and `y` and their Bézout coefficients, i.e. the

integer coefficients u and v that satisfy $ux + vy = d = gcd(x, y)$. $gcdx(x, y)$ returns (d, u, v).

The arguments may be integer and rational numbers.

> **Julia 1.4**
>
> Rational arguments require Julia 1.4 or later.

Examples

```
julia> gcdx(12, 42)
(6, -3, 1)

julia> gcdx(240, 46)
(2, -9, 47)
```

> **Note**
>
> Bézout coefficients are *not* uniquely defined. `gcdx` returns the minimal Bézout coefficients that are computed by the extended Euclidean algorithm. (Ref: D. Knuth, TAoCP, 2/e, p. 325, Algorithm X.) For signed integers, these coefficients `u` and `v` are minimal in the sense that $|u| < |y/d|$ and $|v| < |x/d|$. Furthermore, the signs of `u` and `v` are chosen so that `d` is positive. For unsigned integers, the coefficients `u` and `v` might be near their `typemax`, and the identity then holds only via the unsigned integers' modulo arithmetic.

`Base.ispow2` — Function

```
ispow2(n::Integer) -> Bool
```

Test whether `n` is a power of two.

Examples

```
julia> ispow2(4)
true
```

```
julia> ispow2(5)
false
```

Base.nextpow — Function

```
nextpow(a, x)
```

The smallest `a^n` not less than `x`, where `n` is a non-negative integer. `a` must be greater than 1, and `x` must be greater than 0.

Examples

```
julia> nextpow(2, 7)
8

julia> nextpow(2, 9)
16

julia> nextpow(5, 20)
25

julia> nextpow(4, 16)
16
```

See also `prevpow`.

Base.prevpow — Function

```
prevpow(a, x)
```

The largest `a^n` not greater than `x`, where `n` is a non-negative integer. `a` must be greater than 1, and `x` must not be less than 1.

Examples

```
julia> prevpow(2, 7)
4
```

```
julia> prevpow(2, 9)
8

julia> prevpow(5, 20)
5

julia> prevpow(4, 16)
16
```

See also `nextpow`.

`Base.nextprod` — Function

```
nextprod([k_1, k_2,...], n)
```

Next integer greater than or equal to `n` that can be written as $\prod k_i^{p_i}$ for integers p_1, p_2, etc.

Examples

```
julia> nextprod([2, 3], 105)
108

julia> 2^2 * 3^3
108
```

`Base.invmod` — Function

```
invmod(x,m)
```

Take the inverse of `x` modulo `m`: `y` such that $xy = 1 \pmod m$, with $div(x, y) = 0$. This is undefined for $m = 0$, or if $gcd(x, m) \neq 1$.

Examples

```
julia> invmod(2,5)
3

julia> invmod(2,3)
```

```
2

julia> invmod(5,6)
5
```

Base.powermod — Function

```
powermod(x::Integer, p::Integer, m)
```

Compute $x^p \pmod m$.

Examples

```
julia> powermod(2, 6, 5)
4

julia> mod(2^6, 5)
4

julia> powermod(5, 2, 20)
5

julia> powermod(5, 2, 19)
6

julia> powermod(5, 3, 19)
11
```

Base.ndigits — Function

```
ndigits(n::Integer; base::Integer=10, pad::Integer=1)
```

Compute the number of digits in integer `n` written in base `base` (`base` must not be in `[-1, 0, 1]`), optionally padded with zeros to a specified size (the result will never be less than `pad`).

Examples

```
julia> ndigits(12345)
```

```
5

julia> ndigits(1022, base=16)
3

julia> string(1022, base=16)
"3fe"

julia> ndigits(123, pad=5)
5
```

Base.widemul — Function

```
widemul(x, y)
```

Multiply `x` and `y`, giving the result as a larger type.

Examples

```
julia> widemul(Float32(3.), 4.)
12.0
```

Base.Math.evalpoly — Function

```
evalpoly(x, p)
```

Evaluate the polynomial $\sum_k x^{k-1} p[k]$ for the coefficients `p[1]`, `p[2]`, ...; that is, the coefficients are given in ascending order by power of `x`. Loops are unrolled at compile time if the number of coefficients is statically known, i.e. when `p` is a `Tuple`. This function generates efficient code using Horner's method if `x` is real, or using a Goertzel-like [DK62] algorithm if `x` is complex.

> **Julia 1.4**
>
> This function requires Julia 1.4 or later.

Example

```
julia> evalpoly(2, (1, 2, 3))
17
```

Base.Math.@evalpoly — Macro

```
@evalpoly(z, c...)
```

Evaluate the polynomial $\sum_k z^{k-1} c[k]$ for the coefficients `c[1]`, `c[2]`, ...; that is, the coefficients are given in ascending order by power of `z`. This macro expands to efficient inline code that uses either Horner's method or, for complex `z`, a more efficient Goertzel-like algorithm.

Examples

```
julia> @evalpoly(3, 1, 0, 1)
10

julia> @evalpoly(2, 1, 0, 1)
5

julia> @evalpoly(2, 1, 1, 1)
7
```

Base.FastMath.@fastmath — Macro

```
@fastmath expr
```

Execute a transformed version of the expression, which calls functions that may violate strict IEEE semantics. This allows the fastest possible operation, but results are undefined – be careful when doing this, as it may change numerical results.

This sets the LLVM Fast-Math flags, and corresponds to the `-ffast-math` option in clang. See the notes on performance annotations for more details.

Examples

```
julia> @fastmath 1+2
3
```

```
julia> @fastmath(sin(3))
0.1411200080598672
```

Customizable binary operators

Some unicode characters can be used to define new binary operators that support infix notation. For example `⊗(x,y) = kron(x,y)` defines the `⊗` (otimes) function to be the Kronecker product, and one can call it as binary operator using infix syntax: `C = A ⊗ B` as well as with the usual prefix syntax `C = ⊗(A,B)`.

Other characters that support such extensions include \odot `⊙` and \oplus `⊕`

The complete list is in the parser code: https://github.com/JuliaLang/julia/blob/master/src/julia-parser.scm

Those that are parsed like `*` (in terms of precedence) include `* / ÷ % & ⋅ ∘ × |\\| ∩ ∧ ⊗ ⊘ ⊙ ⊚ ⊛ ⊠ ⊡ ⊓ ∗ ∙ ∤ ⅋ ≀ ⊼ ⋄ ⋆ ⋇ ⋉ ⋊ ⋋ ⋌ ⋏ ⋒ ⟑ ⦸ ⦼ ⦾ ⦿ ⧶ ⧷ ⨇ ⨰ ⨱ ⨲ ⨳ ⨴ ⨵ ⨶ ⨷ ⨸ ⨻ ⨼ ⨽ ⩀ ⩃ ⩄ ⩋ ⩍ ⩎ ⩑ ⩓ ⩕ ⩘ ⩚ ⩜ ⩞ ⩟ ⩠ ⫛ ⊍ ▷ ⨝ ⟕ ⟖ ⟗` and those that are parsed like `+` include `+ - |\|| ⊕ ⊖ ⊞ ⊟ |++| ∪ ∨ ⊔ ± ∓ ∔ ∸ ≏ ⊎ ⊻ ⊽ ⋎ ⋓ ⧺ ⧻ ⨈ ⨢ ⨣ ⨤ ⨥ ⨦ ⨧ ⨨ ⨩ ⨪ ⨫ ⨬ ⨭ ⨮ ⨹ ⨺ ⩁ ⩂ ⩅ ⩊ ⩌ ⩏ ⩐ ⩒ ⩔ ⩖ ⩗ ⩛ ⩝ ⩡ ⩢ ⩣` There are many others that are related to arrows, comparisons, and powers.

- DK62 Donald Knuth, Art of Computer Programming, Volume 2: Seminumerical Algorithms, Sec. 4.6.4.

« Collections and Data Structures Numbers »