Numbers

Standard Numeric Types

Abstract number types

`Core.Number` — Type

```
Number
```

Abstract supertype for all number types.

`Core.Real` — Type

```
Real <: Number
```

Abstract supertype for all real numbers.

`Core.AbstractFloat` — Type

```
AbstractFloat <: Real
```

Abstract supertype for all floating point numbers.

`Core.Integer` — Type

```
Integer <: Real
```

Abstract supertype for all integers.

Core.Signed — Type

```
Signed <: Integer
```

Abstract supertype for all signed integers.

Core.Unsigned — Type

```
Unsigned <: Integer
```

Abstract supertype for all unsigned integers.

Base.AbstractIrrational — Type

```
AbstractIrrational <: Real
```

Number type representing an exact irrational value, which is automatically rounded to the correct precision in arithmetic operations with other numeric quantities.

Subtypes `MyIrrational <: AbstractIrrational` should implement at least `==(::MyIrrational, ::MyIrrational)`, `hash(x::MyIrrational, h::UInt)`, and `convert(::Type{F}, x::MyIrrational) where {F <: Union{BigFloat,Float32,Float64}}`.

If a subtype is used to represent values that may occasionally be rational (e.g. a square-root type that represents √n for integers `n` will give a rational result when `n` is a perfect square), then it should also implement `isinteger`, `iszero`, `isone`, and `==` with `Real` values (since all of these default to `false` for `AbstractIrrational` types), as well as defining `hash` to equal that of the corresponding `Rational`.

Concrete number types

`Core.Float16` — Type

```
Float16 <: AbstractFloat
```

16-bit floating point number type (IEEE 754 standard).

Binary format: 1 sign, 5 exponent, 10 fraction bits.

`Core.Float32` — Type

```
Float32 <: AbstractFloat
```

32-bit floating point number type (IEEE 754 standard).

Binary format: 1 sign, 8 exponent, 23 fraction bits.

`Core.Float64` — Type

```
Float64 <: AbstractFloat
```

64-bit floating point number type (IEEE 754 standard).

Binary format: 1 sign, 11 exponent, 52 fraction bits.

`Base.MPFR.BigFloat` — Type

```
BigFloat <: AbstractFloat
```

Arbitrary precision floating point number type.

`Core.Bool` — Type

```
Bool <: Integer
```

Boolean type, containing the values `true` and `false`.

`Bool` is a kind of number: `false` is numerically equal to `0` and `true` is numerically equal to `1`. Moreover, `false` acts as a multiplicative "strong zero":

```
julia> false == 0
true

julia> true == 1
true

julia> 0 * NaN
NaN

julia> false * NaN
0.0
```

`Core.Int8` — Type

```
Int8 <: Signed
```

8-bit signed integer type.

`Core.UInt8` — Type

```
UInt8 <: Unsigned
```

8-bit unsigned integer type.

`Core.Int16` — Type

```
Int16 <: Signed
```

16-bit signed integer type.

`Core.UInt16` — Type

```
UInt16 <: Unsigned
```

16-bit unsigned integer type.

`Core.Int32` — Type

```
Int32 <: Signed
```

32-bit signed integer type.

`Core.UInt32` — Type

```
UInt32 <: Unsigned
```

32-bit unsigned integer type.

`Core.Int64` — Type

```
Int64 <: Signed
```

64-bit signed integer type.

`Core.UInt64` — Type

```
UInt64 <: Unsigned
```

64-bit unsigned integer type.

`Core.Int128` — Type

```
Int128 <: Signed
```

128-bit signed integer type.

`Core.UInt128` — Type

```
UInt128 <: Unsigned
```

128-bit unsigned integer type.

`Base.GMP.BigInt` — Type

```
BigInt <: Signed
```

Arbitrary precision integer type.

`Base.Complex` — Type

```
Complex{T<:Real} <: Number
```

Complex number type with real and imaginary part of type `T`.

`ComplexF16`, `ComplexF32` and `ComplexF64` are aliases for `Complex{Float16}`, `Complex{Float32}` and `Complex{Float64}` respectively.

`Base.Rational` — Type

```
Rational{T<:Integer} <: Real
```

Rational number type, with numerator and denominator of type `T`. Rationals are checked for overflow.

`Base.Irrational` — Type

```
Irrational{sym} <: AbstractIrrational
```

Number type representing an exact irrational value denoted by the symbol `sym`.

Data Formats

`Base.digits` — Function

```
digits([T<:Integer], n::Integer; base::T = 10, pad::Integer = 1)
```

Return an array with element type `T` (default `Int`) of the digits of `n` in the given base, optionally padded with zeros to a specified size. More significant digits are at higher indices, such that `n == sum([digits[k]*base^(k-1) for k=1:length(digits)])`.

Examples

```
julia> digits(10, base = 10)
2-element Array{Int64,1}:
 0
 1

julia> digits(10, base = 2)
4-element Array{Int64,1}:
 0
 1
 0
 1

julia> digits(10, base = 2, pad = 6)
6-element Array{Int64,1}:
```

```
 0
 1
 0
 1
 0
 0
```

`Base.digits!` — Function

```
digits!(array, n::Integer; base::Integer = 10)
```

Fills an array of the digits of `n` in the given base. More significant digits are at higher indices. If the array length is insufficient, the least significant digits are filled up to the array length. If the array length is excessive, the excess portion is filled with zeros.

Examples

```
julia> digits!([2,2,2,2], 10, base = 2)
4-element Array{Int64,1}:
 0
 1
 0
 1

julia> digits!([2,2,2,2,2,2], 10, base = 2)
6-element Array{Int64,1}:
 0
 1
 0
 1
 0
 0
```

`Base.bitstring` — Function

```
bitstring(n)
```

A string giving the literal bit representation of a number.

Examples

```
julia> bitstring(4)
"0000000000000000000000000000000000000000000000000000000000000100"

julia> bitstring(2.2)
"0100000000000001100110011001100110011001100110011001100110011010"
```

`Base.parse` — Function

```
parse(type, str; base)
```

Parse a string as a number. For `Integer` types, a base can be specified (the default is 10). For floating-point types, the string is parsed as a decimal floating-point number. `Complex` types are parsed from decimal strings of the form `"R±Iim"` as a `Complex(R,I)` of the requested type; `"i"` or `"j"` can also be used instead of `"im"`, and `"R"` or `"Iim"` are also permitted. If the string does not contain a valid number, an error is raised.

> **Julia 1.1**
>
> `parse(Bool, str)` requires at least Julia 1.1.

Examples

```
julia> parse(Int, "1234")
1234

julia> parse(Int, "1234", base = 5)
194

julia> parse(Int, "afc", base = 16)
2812

julia> parse(Float64, "1.2e-3")
0.0012

julia> parse(Complex{Float64}, "3.2e-1 + 4.5im")
0.32 + 4.5im
```

`Base.tryparse` — Function

```
tryparse(type, str; base)
```

Like `parse`, but returns either a value of the requested type, or `nothing` if the string does not contain a valid number.

`Base.big` — Function

```
big(x)
```

Convert a number to a maximum precision representation (typically `BigInt` or `BigFloat`). See `BigFloat` for information about some pitfalls with floating-point numbers.

`Base.signed` — Function

```
signed(T::Integer)
```

Convert an integer bitstype to the signed type of the same size.

Examples

```
julia> signed(UInt16)
Int16
julia> signed(UInt64)
Int64
```

```
signed(x)
```

Convert a number to a signed integer. If the argument is unsigned, it is reinterpreted as signed without checking for overflow.

`Base.unsigned` — Function

```
unsigned(T::Integer)
```

Convert an integer bitstype to the unsigned type of the same size.

Examples

```
julia> unsigned(Int16)
UInt16
```

```
julia> unsigned(UInt64)
UInt64
```

Base.float — Method

```
float(x)
```

Convert a number or array to a floating point data type.

Base.Math.significand — Function

```
significand(x)
```

Extract the `significand(s)` (a.k.a. mantissa), in binary representation, of a floating-point number. If `x` is a non-zero finite number, then the result will be a number of the same type on the interval $[1, 2)$. Otherwise `x` is returned.

Examples

```
julia> significand(15.2)/15.2
0.125

julia> significand(15.2)*8
15.2
```

Base.Math.exponent — Function

```
exponent(x) -> Int
```

Get the exponent of a normalized floating-point number.

Base.complex — Method

```
complex(r, [i])
```

Convert real numbers or arrays to complex. `i` defaults to zero.

Examples

```
julia> complex(7)
7 + 0im

julia> complex([1, 2, 3])
3-element Array{Complex{Int64},1}:
 1 + 0im
 2 + 0im
 3 + 0im
```

`Base.bswap` — Function

```
bswap(n)
```

Reverse the byte order of `n`.

(See also `ntoh` and `hton` to convert between the current native byte order and big-endian order.)

Examples

```
julia> a = bswap(0x10203040)
0x40302010

julia> bswap(a)
0x10203040

julia> string(1, base = 2)
"1"

julia> string(bswap(1), base = 2)
"10000000000000000000000000000000000000000000000000000000"
```

`Base.hex2bytes` — Function

```
hex2bytes(s::Union{AbstractString,AbstractVector{UInt8}})
```

Given a string or array `s` of ASCII codes for a sequence of hexadecimal digits, returns a `Vector{UInt8}` of bytes corresponding to the binary representation: each successive pair of hexadecimal digits in `s` gives the value of one byte in the return vector.

The length of `s` must be even, and the returned array has half of the length of `s`. See also `hex2bytes!` for an in-place version, and `bytes2hex` for the inverse.

Examples

```
julia> s = string(12345, base = 16)
"3039"

julia> hex2bytes(s)
2-element Array{UInt8,1}:
 0x30
 0x39

julia> a = b"01abEF"
6-element Base.CodeUnits{UInt8,String}:
 0x30
 0x31
 0x61
 0x62
 0x45
 0x46

julia> hex2bytes(a)
3-element Array{UInt8,1}:
 0x01
 0xab
 0xef
```

`Base.hex2bytes!` — Function

```
hex2bytes!(d::AbstractVector{UInt8}, s::Union{String,AbstractVector{UInt8}})
```

Convert an array `s` of bytes representing a hexadecimal string to its binary representation, similar to `hex2bytes` except that the output is written in-place in `d`. The length of `s` must be exactly twice

the length of `d`.

`Base.bytes2hex` — Function

```
bytes2hex(a::AbstractArray{UInt8}) -> String
bytes2hex(io::IO, a::AbstractArray{UInt8})
```

Convert an array `a` of bytes to its hexadecimal string representation, either returning a `String` via `bytes2hex(a)` or writing the string to an `io` stream via `bytes2hex(io, a)`. The hexadecimal characters are all lowercase.

Examples

```
julia> a = string(12345, base = 16)
"3039"

julia> b = hex2bytes(a)
2-element Array{UInt8,1}:
 0x30
 0x39

julia> bytes2hex(b)
"3039"
```

General Number Functions and Constants

`Base.one` — Function

```
one(x)
one(T::type)
```

Return a multiplicative identity for `x`: a value such that `one(x)*x == x*one(x) == x`. Alternatively `one(T)` can take a type `T`, in which case `one` returns a multiplicative identity for any `x` of type `T`.

If possible, `one(x)` returns a value of the same type as `x`, and `one(T)` returns a value of type `T`. However, this may not be the case for types representing dimensionful quantities (e.g. time in

days), since the multiplicative identity must be dimensionless. In that case, `one(x)` should return an identity value of the same precision (and shape, for matrices) as `x`.

If you want a quantity that is of the same type as `x`, or of type `T`, even if `x` is dimensionful, use `oneunit` instead.

Examples

```
julia> one(3.7)
1.0

julia> one(Int)
1

julia> import Dates; one(Dates.Day(1))
1
```

`Base.oneunit` — Function

```
oneunit(x::T)
oneunit(T::Type)
```

Returns `T(one(x))`, where `T` is either the type of the argument or (if a type is passed) the argument. This differs from `one` for dimensionful quantities: `one` is dimensionless (a multiplicative identity) while `oneunit` is dimensionful (of the same type as `x`, or of type `T`).

Examples

```
julia> oneunit(3.7)
1.0

julia> import Dates; oneunit(Dates.Day)
1 day
```

`Base.zero` — Function

```
zero(x)
```

Get the additive identity element for the type of `x` (`x` can also specify the type itself).

Examples

```
julia> zero(1)
0

julia> zero(big"2.0")
0.0

julia> zero(rand(2,2))
2×2 Array{Float64,2}:
 0.0  0.0
 0.0  0.0
```

`Base.im` — Constant

```
im
```

The imaginary unit.

Examples

```
julia> im * im
-1 + 0im
```

`Base.MathConstants.pi` — Constant

```
π
pi
```

The constant pi.

Examples

```
julia> pi
π = 3.1415926535897...
```

Base.MathConstants.ℯ — Constant

```
ℯ
e
```

The constant e.

Examples

```
julia> ℯ
ℯ = 2.7182818284590...
```

Base.MathConstants.catalan — Constant

```
catalan
```

Catalan's constant.

Examples

```
julia> Base.MathConstants.catalan
catalan = 0.9159655941772...
```

Base.MathConstants.eulergamma — Constant

```
γ
eulergamma
```

Euler's constant.

Examples

```
julia> Base.MathConstants.eulergamma
γ = 0.5772156649015...
```

Base.MathConstants.golden — Constant

```
φ
golden
```

The golden ratio.

Examples

```
julia> Base.MathConstants.golden
φ = 1.6180339887498...
```

Base.Inf — Constant

```
Inf, Inf64
```

Positive infinity of type `Float64`.

Base.Inf32 — Constant

```
Inf32
```

Positive infinity of type `Float32`.

Base.Inf16 — Constant

```
Inf16
```

Positive infinity of type `Float16`.

Base.NaN — Constant

```
NaN, NaN64
```

A not-a-number value of type `Float64`.

`Base.NaN32` — Constant

```
NaN32
```

A not-a-number value of type `Float32`.

`Base.NaN16` — Constant

```
NaN16
```

A not-a-number value of type `Float16`.

`Base.issubnormal` — Function

```
issubnormal(f) -> Bool
```

Test whether a floating point number is subnormal.

`Base.isfinite` — Function

```
isfinite(f) -> Bool
```

Test whether a number is finite.

Examples

```
julia> isfinite(5)
true
```

```
julia> isfinite(NaN32)
false
```

Base.isinf — Function

```
isinf(f) -> Bool
```

Test whether a number is infinite.

Base.isnan — Function

```
isnan(f) -> Bool
```

Test whether a number value is a NaN, an indeterminate value which is neither an infinity nor a finite number ("not a number").

Base.iszero — Function

```
iszero(x)
```

Return `true` if `x == zero(x)`; if `x` is an array, this checks whether all of the elements of `x` are zero.

Examples

```
julia> iszero(0.0)
true

julia> iszero([1, 9, 0])
false

julia> iszero([false, 0, 0])
true
```

Base.isone — Function

```
isone(x)
```

Return `true` if `x == one(x)`; if `x` is an array, this checks whether `x` is an identity matrix.

Examples

```
julia> isone(1.0)
true

julia> isone([1 0; 0 2])
false

julia> isone([1 0; 0 true])
true
```

Base.nextfloat — Function

```
nextfloat(x::AbstractFloat, n::Integer)
```

The result of `n` iterative applications of `nextfloat` to `x` if `n >= 0`, or `-n` applications of `prevfloat` if `n < 0`.

```
nextfloat(x::AbstractFloat)
```

Return the smallest floating point number `y` of the same type as `x` such `x < y`. If no such `y` exists (e.g. if `x` is `Inf` or `NaN`), then return `x`.

Base.prevfloat — Function

```
prevfloat(x::AbstractFloat, n::Integer)
```

The result of `n` iterative applications of `prevfloat` to `x` if `n >= 0`, or `-n` applications of `nextfloat` if `n < 0`.

```
prevfloat(x::AbstractFloat)
```

Return the largest floating point number `y` of the same type as `x` such `y < x`. If no such `y` exists (e.g. if `x` is `-Inf` or `NaN`), then return `x`.

`Base.isinteger` — Function

```
isinteger(x) -> Bool
```

Test whether `x` is numerically equal to some integer.

Examples

```
julia> isinteger(4.0)
true
```

`Base.isreal` — Function

```
isreal(x) -> Bool
```

Test whether `x` or all its elements are numerically equal to some real number including infinities and NaNs. `isreal(x)` is true if `isequal(x, real(x))` is true.

Examples

```
julia> isreal(5.)
true

julia> isreal(Inf + 0im)
true
```

```
julia> isreal([4.; complex(0,1)])
false
```

Core.Float32 — Method

```
Float32(x [, mode::RoundingMode])
```

Create a `Float32` from `x`. If `x` is not exactly representable then `mode` determines how `x` is rounded.

Examples

```
julia> Float32(1/3, RoundDown)
0.3333333f0

julia> Float32(1/3, RoundUp)
0.33333334f0
```

See `RoundingMode` for available rounding modes.

Core.Float64 — Method

```
Float64(x [, mode::RoundingMode])
```

Create a `Float64` from `x`. If `x` is not exactly representable then `mode` determines how `x` is rounded.

Examples

```
julia> Float64(pi, RoundDown)
3.141592653589793

julia> Float64(pi, RoundUp)
3.1415926535897936
```

See `RoundingMode` for available rounding modes.

`Base.Rounding.rounding` — Function

```
rounding(T)
```

Get the current floating point rounding mode for type `T`, controlling the rounding of basic arithmetic functions (`+`, `-`, `*`, `/` and `sqrt`) and type conversion.

See `RoundingMode` for available modes.

`Base.Rounding.setrounding` — Method

```
setrounding(T, mode)
```

Set the rounding mode of floating point type `T`, controlling the rounding of basic arithmetic functions (`+`, `-`, `*`, `/` and `sqrt`) and type conversion. Other numerical functions may give incorrect or invalid values when using rounding modes other than the default `RoundNearest`.

Note that this is currently only supported for `T == BigFloat`.

Warning

This function is not thread-safe. It will affect code running on all threads, but its behavior is undefined if called concurrently with computations that use the setting.

`Base.Rounding.setrounding` — Method

```
setrounding(f::Function, T, mode)
```

Change the rounding mode of floating point type `T` for the duration of `f`. It is logically equivalent to:

```
old = rounding(T)
setrounding(T, mode)
f()
setrounding(T, old)
```

See `RoundingMode` for available rounding modes.

`Base.Rounding.get_zero_subnormals` — Function

```
get_zero_subnormals() -> Bool
```

Return `false` if operations on subnormal floating-point values ("denormals") obey rules for IEEE arithmetic, and `true` if they might be converted to zeros.

> **Warning**
>
> This function only affects the current thread.

`Base.Rounding.set_zero_subnormals` — Function

```
set_zero_subnormals(yes::Bool) -> Bool
```

If `yes` is `false`, subsequent floating-point operations follow rules for IEEE arithmetic on subnormal values ("denormals"). Otherwise, floating-point operations are permitted (but not required) to convert subnormal inputs or outputs to zero. Returns `true` unless `yes==true` but the hardware does not support zeroing of subnormal numbers.

`set_zero_subnormals(true)` can speed up some computations on some hardware. However, it can break identities such as `(x-y==0) == (x==y)`.

> **Warning**
>
> This function only affects the current thread.

Integers

`Base.count_ones` — Function

```
count_ones(x::Integer) -> Integer
```

Number of ones in the binary representation of `x`.

Examples

```
julia> count_ones(7)
3
```

`Base.count_zeros` — Function

```
count_zeros(x::Integer) -> Integer
```

Number of zeros in the binary representation of `x`.

Examples

```
julia> count_zeros(Int32(2 ^ 16 - 1))
16
```

`Base.leading_zeros` — Function

```
leading_zeros(x::Integer) -> Integer
```

Number of zeros leading the binary representation of `x`.

Examples

```
julia> leading_zeros(Int32(1))
31
```

`Base.leading_ones` — Function

```
leading_ones(x::Integer) -> Integer
```

Number of ones leading the binary representation of `x`.

Examples

```
julia> leading_ones(UInt32(2 ^ 32 - 2))
31
```

`Base.trailing_zeros` — Function

```
trailing_zeros(x::Integer) -> Integer
```

Number of zeros trailing the binary representation of `x`.

Examples

```
julia> trailing_zeros(2)
1
```

`Base.trailing_ones` — Function

```
trailing_ones(x::Integer) -> Integer
```

Number of ones trailing the binary representation of `x`.

Examples

```
julia> trailing_ones(3)
2
```

`Base.isodd` — Function

```
isodd(x::Integer) -> Bool
```

Return `true` if `x` is odd (that is, not divisible by 2), and `false` otherwise.

Examples

```
julia> isodd(9)
true

julia> isodd(10)
false
```

`Base.iseven` — Function

```
iseven(x::Integer) -> Bool
```

Return `true` if `x` is even (that is, divisible by 2), and `false` otherwise.

Examples

```
julia> iseven(9)
false

julia> iseven(10)
true
```

`Core.@int128_str` — Macro

```
@int128_str str
@int128_str(str)
```

`@int128_str` parses a string into a Int128 Throws an `ArgumentError` if the string is not a valid integer

`Core.@uint128_str` — Macro

```
@uint128_str str
@uint128_str(str)
```

`@uint128_str` parses a string into a UInt128 Throws an `ArgumentError` if the string is not a valid integer

BigFloats and BigInts

The `BigFloat` and `BigInt` types implements arbitrary-precision floating point and integer arithmetic, respectively. For `BigFloat` the GNU MPFR library is used, and for `BigInt` the GNU Multiple Precision Arithmetic Library (GMP) is used.

`Base.MPFR.BigFloat` — Method

```
BigFloat(x::Union{Real, AbstractString} [, rounding::RoundingMode=rounding(BigF
```

Create an arbitrary precision floating point number from `x`, with precision `precision`. The `rounding` argument specifies the direction in which the result should be rounded if the conversion cannot be done exactly. If not provided, these are set by the current global values.

`BigFloat(x::Real)` is the same as `convert(BigFloat,x)`, except if `x` itself is already `BigFloat`, in which case it will return a value with the precision set to the current global precision; `convert` will always return `x`.

`BigFloat(x::AbstractString)` is identical to `parse`. This is provided for convenience since decimal literals are converted to `Float64` when parsed, so `BigFloat(2.1)` may not yield what you expect.

Julia 1.1

`precision` as a keyword argument requires at least Julia 1.1. In Julia 1.0 `precision` is the second positional argument (`BigFloat(x, precision)`).

Examples

```
julia> BigFloat(2.1) # 2.1 here is a Float64
2.100000000000000088817841970012523233890533447265625
```

```
julia> BigFloat("2.1") # the closest BigFloat to 2.1
2.099999999999999999999999999999999999999999999999999999999999999999999999999998

julia> BigFloat("2.1", RoundUp)
2.100000000000000000000000000000000000000000000000000000000000000000000000000002

julia> BigFloat("2.1", RoundUp, precision=128)
2.100000000000000000000000000000000007
```

See also

- `@big_str`
- `rounding` and `setrounding`
- `precision` and `setprecision`

`Base.precision` — Function

```
precision(num::AbstractFloat)
```

Get the precision of a floating point number, as defined by the effective number of bits in the significand.

`Base.precision` — Method

```
precision(BigFloat)
```

Get the precision (in bits) currently used for `BigFloat` arithmetic.

`Base.MPFR.setprecision` — Function

```
setprecision([T=BigFloat,] precision::Int)
```

Set the precision (in bits) to be used for `T` arithmetic.

> **Warning**
>
> This function is not thread-safe. It will affect code running on all threads, but its behavior is undefined if called concurrently with computations that use the setting.

```
setprecision(f::Function, [T=BigFloat,] precision::Integer)
```

Change the `T` arithmetic precision (in bits) for the duration of `f`. It is logically equivalent to:

```
old = precision(BigFloat)
setprecision(BigFloat, precision)
f()
setprecision(BigFloat, old)
```

Often used as `setprecision(T, precision) do ... end`

Note: `nextfloat()`, `prevfloat()` do not use the precision mentioned by `setprecision`

`Base.GMP.BigInt` — Method

```
BigInt(x)
```

Create an arbitrary precision integer. `x` may be an `Int` (or anything that can be converted to an `Int`). The usual mathematical operators are defined for this type, and results are promoted to a `BigInt`.

Instances can be constructed from strings via `parse`, or using the `big` string literal.

Examples

```
julia> parse(BigInt, "42")
42

julia> big"313"
313

julia> BigInt(10)^19
10000000000000000000
```

`Core.@big_str` — Macro

```
@big_str str
@big_str(str)
```

Parse a string into a `BigInt` or `BigFloat`, and throw an `ArgumentError` if the string is not a valid number. For integers `_` is allowed in the string as a separator.

Examples

```
julia> big"123_456"
123456

julia> big"7891.5"
7891.5
```

« Mathematics Strings »