Essentials

Introduction

Julia Base contains a range of functions and macros appropriate for performing scientific and numerical computing, but is also as broad as those of many general purpose programming languages. Additional functionality is available from a growing collection of available packages. Functions are grouped by topic below.

Some general notes:

- To use module functions, use `import Module` to import the module, and `Module.fn(x)` to use the functions.
- Alternatively, `using Module` will import all exported `Module` functions into the current namespace.
- By convention, function names ending with an exclamation point (`!`) modify their arguments. Some functions have both modifying (e.g., `sort!`) and non-modifying (`sort`) versions.

Getting Around

`Base.exit` — Function

```
exit(code=0)
```

Stop the program with an exit code. The default exit code is zero, indicating that the program completed successfully. In an interactive session, `exit()` can be called with the keyboard shortcut `^D`.

`Base.atexit` — Function

```
atexit(f)
```

Register a zero-argument function `f()` to be called at process exit. `atexit()` hooks are called in last in first out (LIFO) order and run before object finalizers.

Exit hooks are allowed to call `exit(n)`, in which case Julia will exit with exit code `n` (instead of the original exit code). If more than one exit hook calls `exit(n)`, then Julia will exit with the exit code corresponding to the last called exit hook that calls `exit(n)`. (Because exit hooks are called in LIFO order, "last called" is equivalent to "first registered".)

`Base.isinteractive` — Function

```
isinteractive() -> Bool
```

Determine whether Julia is running an interactive session.

`Base.summarysize` — Function

```
Base.summarysize(obj; exclude=Union{...}, chargeall=Union{...}) -> Int
```

Compute the amount of memory, in bytes, used by all unique objects reachable from the argument.

Keyword Arguments

- `exclude`: specifies the types of objects to exclude from the traversal.
- `chargeall`: specifies the types of objects to always charge the size of all of their fields, even if those fields would normally be excluded.

`Base.require` — Function

```
require(into::Module, module::Symbol)
```

This function is part of the implementation of `using` / `import`, if a module is not already defined in `Main`. It can also be called directly to force reloading a module, regardless of whether it has been loaded before (for example, when interactively developing libraries).

Loads a source file, in the context of the `Main` module, on every active node, searching standard locations for files. `require` is considered a top-level operation, so it sets the current `include` path but does not use it to search for files (see help for `include`). This function is typically used to load library code, and is implicitly called by `using` to load packages.

When searching for files, `require` first looks for package code in the global array `LOAD_PATH`. `require` is case-sensitive on all platforms, including those with case-insensitive filesystems like macOS and Windows.

For more details regarding code loading, see the manual sections on modules and parallel computing.

`Base.compilecache` — Function

```
Base.compilecache(module::PkgId)
```

Creates a precompiled cache file for a module and all of its dependencies. This can be used to reduce package load times. Cache files are stored in `DEPOT_PATH[1]/compiled`. See Module initialization and precompilation for important notes.

`Base.__precompile__` — Function

```
__precompile__(isprecompilable::Bool)
```

Specify whether the file calling this function is precompilable, defaulting to `true`. If a module or file is *not* safely precompilable, it should call `__precompile__(false)` in order to throw an error if Julia attempts to precompile it.

`Base.include` — Function

```
Base.include([mapexpr::Function,] [m::Module,] path::AbstractString)
```

Evaluate the contents of the input source file in the global scope of module `m`. Every module (except those defined with `baremodule`) has its own definition of `include` omitting the `m` argument, which evaluates the file in that module. Returns the result of the last evaluated expression of the input file. During including, a task-local include path is set to the directory containing the file. Nested calls to `include` will search relative to that path. This function is typically used to load source interactively, or to combine files in packages that are broken into multiple source files.

The optional first argument `mapexpr` can be used to transform the included code before it is

evaluated: for each parsed expression `expr` in `path`, the `include` function actually evaluates `mapexpr(expr)`. If it is omitted, `mapexpr` defaults to `identity`.

`Base.MainInclude.include` — Function

```
include([mapexpr::Function,] path::AbstractString)
```

Evaluate the contents of the input source file in the global scope of the containing module. Every module (except those defined with `baremodule`) has its own definition of `include`, which evaluates the file in that module. Returns the result of the last evaluated expression of the input file. During including, a task-local include path is set to the directory containing the file. Nested calls to `include` will search relative to that path. This function is typically used to load source interactively, or to combine files in packages that are broken into multiple source files.

The optional first argument `mapexpr` can be used to transform the included code before it is evaluated: for each parsed expression `expr` in `path`, the `include` function actually evaluates `mapexpr(expr)`. If it is omitted, `mapexpr` defaults to `identity`.

Use `Base.include` to evaluate a file into another module.

`Base.include_string` — Function

```
include_string([mapexpr::Function,] m::Module, code::AbstractString, filename::
```

Like `include`, except reads code from the given string rather than from a file.

The optional first argument `mapexpr` can be used to transform the included code before it is evaluated: for each parsed expression `expr` in `code`, the `include_string` function actually evaluates `mapexpr(expr)`. If it is omitted, `mapexpr` defaults to `identity`.

`Base.include_dependency` — Function

```
include_dependency(path::AbstractString)
```

In a module, declare that the file specified by `path` (relative or absolute) is a dependency for precompilation; that is, the module will need to be recompiled if this file changes.

This is only needed if your module depends on a file that is not used via `include`. It has no effect outside of compilation.

`Base.which` — Method

```
which(f, types)
```

Returns the method of `f` (a `Method` object) that would be called for arguments of the given `types`.

If `types` is an abstract type, then the method that would be called by `invoke` is returned.

`Base.methods` — Function

```
methods(f, [types], [module])
```

Return the method table for `f`.

If `types` is specified, return an array of methods whose types match. If `module` is specified, return an array of methods defined in that module. A list of modules can also be specified as an array.

> **Julia 1.4**
>
> At least Julia 1.4 is required for specifying a module.

`Base.@show` — Macro

```
@show
```

Show an expression and result, returning the result. See also `show`.

`ans` — Keyword

```
ans
```

A variable referring to the last computed value, automatically set at the interactive prompt.

Keywords

This is the list of reserved keywords in Julia: `baremodule`, `begin`, `break`, `catch`, `const`, `continue`, `do`, `else`, `elseif`, `end`, `export`, `false`, `finally`, `for`, `function`, `global`, `if`, `import`, `let`, `local`, `macro`, `module`, `quote`, `return`, `struct`, `true`, `try`, `using`, `while`. Those keywords are not allowed to be used as variable names.

The following two-word sequences are reserved: `abstract type`, `mutable struct`, `primitive type`. However, you can create variables with names: `abstract`, `mutable`, `primitive` and `type`.

Finally, `where` is parsed as an infix operator for writing parametric method and type definitions. Also `in` and `isa` are parsed as infix operators. Creation of a variable named `where`, `in` or `isa` is allowed though.

`module` — Keyword

```
module
```

`module` declares a `Module`, which is a separate global variable workspace. Within a module, you can control which names from other modules are visible (via importing), and specify which of your names are intended to be public (via exporting). Modules allow you to create top-level definitions without worrying about name conflicts when your code is used together with somebody else's. See the manual section about modules for more details.

Examples

```
module Foo
import Base.show
export MyType, foo

struct MyType
    x
end

bar(x) = 2x
foo(a::MyType) = bar(a.x) + 1
show(io::IO, a::MyType) = print(io, "MyType $(a.x)")
end
```

`export` — Keyword

```
export
```

`export` is used within modules to tell Julia which functions should be made available to the user. For example: `export foo` makes the name `foo` available when `using` the module. See the manual section about modules for details.

`import` — Keyword

```
import
```

`import Foo` will load the module or package `Foo`. Names from the imported `Foo` module can be accessed with dot syntax (e.g. `Foo.foo` to access the name `foo`). See the manual section about modules for details.

`using` — Keyword

```
using
```

`using Foo` will load the module or package `Foo` and make its `export`ed names available for direct use. Names can also be used via dot syntax (e.g. `Foo.foo` to access the name `foo`), whether they are `export`ed or not. See the manual section about modules for details.

`baremodule` — Keyword

```
baremodule
```

`baremodule` declares a module that does not contain `using Base` or a definition of `eval`. It does still import `Core`.

`function` — Keyword

```
function
```

Functions are defined with the `function` keyword:

```
function add(a, b)
    return a + b
end
```

Or the short form notation:

```
add(a, b) = a + b
```

The use of the `return` keyword is exactly the same as in other languages, but is often optional. A function without an explicit `return` statement will return the last expression in the function body.

`macro` — Keyword

```
macro
```

`macro` defines a method for inserting generated code into a program. A macro maps a sequence of argument expressions to a returned expression, and the resulting expression is substituted directly into the program at the point where the macro is invoked. Macros are a way to run generated code without calling `eval`, since the generated code instead simply becomes part of the surrounding program. Macro arguments may include expressions, literal values, and symbols. Macros can be defined for variable number of arguments (varargs), but do not accept keyword arguments.

Examples

```
julia> macro sayhello(name)
           return :( println("Hello, ", $name, "!") )
       end
@sayhello (macro with 1 method)

julia> @sayhello "Charlie"
Hello, Charlie!

julia> macro saylots(x...)
```

```
            return :( println("Say: ", $(x...)) )
        end
@saylots (macro with 1 method)

julia> @saylots "hey " "there " "friend"
Say: hey there friend
```

`return` — Keyword

```
return
```

`return x` causes the enclosing function to exit early, passing the given value `x` back to its caller. `return` by itself with no value is equivalent to `return nothing` (see `nothing`).

```
function compare(a, b)
    a == b && return "equal to"
    a < b ? "less than" : "greater than"
end
```

In general you can place a `return` statement anywhere within a function body, including within deeply nested loops or conditionals, but be careful with `do` blocks. For example:

```
function test1(xs)
    for x in xs
        iseven(x) && return 2x
    end
end

function test2(xs)
    map(xs) do x
        iseven(x) && return 2x
        x
    end
end
```

In the first example, the return breaks out of `test1` as soon as it hits an even number, so `test1([5,6,7])` returns `12`.

You might expect the second example to behave the same way, but in fact the `return` there only breaks out of the *inner* function (inside the `do` block) and gives a value back to `map`.

`test2([5,6,7])` then returns `[5,12,7]`.

When used in a top-level expression (i.e. outside any function), `return` causes the entire current top-level expression to terminate early.

`do` — Keyword

```
do
```

Create an anonymous function and pass it as the first argument to a function call. For example:

```
map(1:10) do x
    2x
end
```

is equivalent to `map(x->2x, 1:10)`.

Use multiple arguments like so:

```
map(1:10, 11:20) do x, y
    x + y
end
```

`begin` — Keyword

```
begin
```

`begin...end` denotes a block of code.

```
begin
    println("Hello, ")
    println("World!")
end
```

Usually `begin` will not be necessary, since keywords such as `function` and `let` implicitly begin blocks of code. See also `;`.

`end` — Keyword

```
end
```

`end` marks the conclusion of a block of expressions, for example `module`, `struct`, `mutable struct`, `begin`, `let`, `for` etc. `end` may also be used when indexing into an array to represent the last index of a dimension.

Examples

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

julia> A[end, :]
2-element Array{Int64,1}:
 3
 4
```

`let` — Keyword

```
let
```

`let` statements allocate new variable bindings each time they run. Whereas an assignment modifies an existing value location, `let` creates new locations. This difference is only detectable in the case of variables that outlive their scope via closures. The `let` syntax accepts a comma-separated series of assignments and variable names:

```
let var1 = value1, var2, var3 = value3
    code
end
```

The assignments are evaluated in order, with each right-hand side evaluated in the scope before the new variable on the left-hand side has been introduced. Therefore it makes sense to write something like `let x = x`, since the two `x` variables are distinct and have separate storage.

`if` — Keyword

```
if/elseif/else
```

`if/elseif/else` performs conditional evaluation, which allows portions of code to be evaluated or not evaluated depending on the value of a boolean expression. Here is the anatomy of the `if/elseif/else` conditional syntax:

```
if x < y
    println("x is less than y")
elseif x > y
    println("x is greater than y")
else
    println("x is equal to y")
end
```

If the condition expression `x < y` is true, then the corresponding block is evaluated; otherwise the condition expression `x > y` is evaluated, and if it is true, the corresponding block is evaluated; if neither expression is true, the `else` block is evaluated. The `elseif` and `else` blocks are optional, and as many `elseif` blocks as desired can be used.

`for` — Keyword

```
for
```

`for` loops repeatedly evaluate a block of statements while iterating over a sequence of values.

Examples

```
julia> for i in [1, 4, 0]
           println(i)
       end
1
4
0
```

`while` — Keyword

```
while
```

`while` loops repeatedly evaluate a conditional expression, and continue evaluating the body of the while loop as long as the expression remains true. If the condition expression is false when the while loop is first reached, the body is never evaluated.

Examples

```
julia> i = 1
1

julia> while i < 5
           println(i)
           global i += 1
       end
1
2
3
4
```

`break` — Keyword

```
break
```

Break out of a loop immediately.

Examples

```
julia> i = 0
0

julia> while true
           global i += 1
           i > 5 && break
           println(i)
       end
1
```

```
2
3
4
5
```

`continue` — Keyword

```
continue
```

Skip the rest of the current loop iteration.

Examples

```
julia> for i = 1:6
           iseven(i) && continue
           println(i)
       end
1
3
5
```

`try` — Keyword

```
try/catch
```

A `try/catch` statement allows intercepting errors (exceptions) thrown by `throw` so that program execution can continue. For example, the following code attempts to write a file, but warns the user and proceeds instead of terminating execution if the file cannot be written:

```
try
    open("/danger", "w") do f
        println(f, "Hello")
    end
catch
    @warn "Could not write file."
end
```

or, when the file cannot be read into a variable:

```
lines = try
    open("/danger", "r") do f
        readlines(f)
    end
catch
    @warn "File not found."
end
```

The syntax `catch e` (where `e` is any variable) assigns the thrown exception object to the given variable within the `catch` block.

The power of the `try/catch` construct lies in the ability to unwind a deeply nested computation immediately to a much higher level in the stack of calling functions.

`finally` — Keyword

```
finally
```

Run some code when a given block of code exits, regardless of how it exits. For example, here is how we can guarantee that an opened file is closed:

```
f = open("file")
try
    operate_on_file(f)
finally
    close(f)
end
```

When control leaves the `try` block (for example, due to a `return`, or just finishing normally), `close(f)` will be executed. If the `try` block exits due to an exception, the exception will continue propagating. A `catch` block may be combined with `try` and `finally` as well. In this case the `finally` block will run after `catch` has handled the error.

`quote` — Keyword

```
quote
```

`quote` creates multiple expression objects in a block without using the explicit `Expr` constructor.

For example:

```
ex = quote
    x = 1
    y = 2
    x + y
end
```

Unlike the other means of quoting, `:( ... )`, this form introduces `QuoteNode` elements to the expression tree, which must be considered when directly manipulating the tree. For other purposes, `:( ... )` and `quote .. end` blocks are treated identically.

`local` — Keyword

```
local
```

`local` introduces a new local variable. See the manual section on variable scoping for more information.

Examples

```
julia> function foo(n)
           x = 0
           for i = 1:n
               local x # introduce a loop-local x
               x = i
           end
           x
       end
foo (generic function with 1 method)

julia> foo(10)
0
```

`global` — Keyword

```
global
```

`global x` makes `x` in the current scope and its inner scopes refer to the global variable of that name. See the manual section on variable scoping for more information.

Examples

```
julia> z = 3
3

julia> function foo()
           global z = 6 # use the z variable defined outside foo
       end
foo (generic function with 1 method)

julia> foo()
6

julia> z
6
```

`const` — Keyword

```
const
```

`const` is used to declare global variables whose values will not change. In almost all code (and particularly performance sensitive code) global variables should be declared constant in this way.

```
const x = 5
```

Multiple variables can be declared within a single `const`:

```
const y, z = 7, 11
```

Note that `const` only applies to one `=` operation, therefore `const x = y = 1` declares `x` to be constant but not `y`. On the other hand, `const x = const y = 1` declares both `x` and `y` constant.

Note that "constant-ness" does not extend into mutable containers; only the association between a variable and its value is constant. If `x` is an array or dictionary (for example) you can still modify, add, or remove elements.

In some cases changing the value of a `const` variable gives a warning instead of an error. However,

this can produce unpredictable behavior or corrupt the state of your program, and so should be avoided. This feature is intended only for convenience during interactive use.

`struct` — Keyword

```
struct
```

The most commonly used kind of type in Julia is a struct, specified as a name and a set of fields.

```
struct Point
    x
    y
end
```

Fields can have type restrictions, which may be parameterized:

```
struct Point{X}
    x::X
    y::Float64
end
```

A struct can also declare an abstract super type via `<:` syntax:

```
struct Point <: AbstractPoint
    x
    y
end
```

`struct`s are immutable by default; an instance of one of these types cannot be modified after construction. Use `mutable struct` instead to declare a type whose instances can be modified.

See the manual section on Composite Types for more details, such as how to define constructors.

`mutable struct` — Keyword

```
mutable struct
```

`mutable struct` is similar to `struct`, but additionally allows the fields of the type to be set after

construction. See the manual section on Composite Types for more information.

`abstract type` — Keyword

```
abstract type
```

`abstract type` declares a type that cannot be instantiated, and serves only as a node in the type graph, thereby describing sets of related concrete types: those concrete types which are their descendants. Abstract types form the conceptual hierarchy which makes Julia's type system more than just a collection of object implementations. For example:

```
abstract type Number end
abstract type Real <: Number end
```

`Number` has no supertype, whereas `Real` is an abstract subtype of `Number`.

`primitive type` — Keyword

```
primitive type
```

`primitive type` declares a concrete type whose data consists only of a series of bits. Classic examples of primitive types are integers and floating-point values. Some example built-in primitive type declarations:

```
primitive type Char 32 end
primitive type Bool <: Integer 8 end
```

The number after the name indicates how many bits of storage the type requires. Currently, only sizes that are multiples of 8 bits are supported. The `Bool` declaration shows how a primitive type can be optionally declared to be a subtype of some supertype.

`where` — Keyword

```
where
```

The `where` keyword creates a type that is an iterated union of other types, over all values of some variable. For example `Vector{T} where T<:Real` includes all `Vector`s where the element type is some kind of `Real` number.

The variable bound defaults to `Any` if it is omitted:

```
Vector{T} where T    # short for `where T<:Any`
```

Variables can also have lower bounds:

```
Vector{T} where T>:Int
Vector{T} where Int<:T<:Real
```

There is also a concise syntax for nested `where` expressions. For example, this:

```
Pair{T, S} where S<:Array{T} where T<:Number
```

can be shortened to:

```
Pair{T, S} where {T<:Number, S<:Array{T}}
```

This form is often found on method signatures.

Note that in this form, the variables are listed outermost-first. This matches the order in which variables are substituted when a type is "applied" to parameter values using the syntax `T{p1, p2, ...}`.

`...` — Keyword

```
...
```

The "splat" operator, `...`, represents a sequence of arguments. `...` can be used in function definitions, to indicate that the function accepts an arbitrary number of arguments. `...` can also be used to apply a function to a sequence of arguments.

Examples

```
julia> add(xs...) = reduce(+, xs)
add (generic function with 1 method)
```

```
julia> add(1, 2, 3, 4, 5)
15

julia> add([1, 2, 3]...)
6

julia> add(7, 1:100..., 1000:1100...)
111107
```

; — Keyword

```
;
```

; has a similar role in Julia as in many C-like languages, and is used to delimit the end of the previous statement. ; is not necessary after new lines, but can be used to separate statements on a single line or to join statements into a single expression. ; is also used to suppress output printing in the REPL and similar interfaces.

Examples

```
julia> function foo()
           x = "Hello, "; x *= "World!"
           return x
       end
foo (generic function with 1 method)

julia> bar() = (x = "Hello, Mars!"; return x)
bar (generic function with 1 method)

julia> foo();

julia> bar()
"Hello, Mars!"
```

= — Keyword

```
=
```

`=` is the assignment operator.

- For variable `a` and expression `b`, `a = b` makes `a` refer to the value of `b`.
- For functions `f(x)`, `f(x) = x` defines a new function constant `f`, or adds a new method to `f` if `f` is already defined; this usage is equivalent to `function f(x); x; end`.
- `a[i] = v` calls `setindex!(a,v,i)`.
- `a.b = c` calls `setproperty!(a,:b,c)`.
- Inside a function call, `f(a=b)` passes `b` as the value of keyword argument `a`.
- Inside parentheses with commas, `(a=1,)` constructs a `NamedTuple`.

Examples

Assigning `a` to `b` does not create a copy of `b`; instead use `copy` or `deepcopy`.

```
julia> b = [1]; a = b; b[1] = 2; a
1-element Array{Int64,1}:
 2

julia> b = [1]; a = copy(b); b[1] = 2; a
1-element Array{Int64,1}:
 1
```

Collections passed to functions are also not copied. Functions can modify (mutate) the contents of the objects their arguments refer to. (The names of functions which do this are conventionally suffixed with '!'.)

```
julia> function f!(x); x[:] .+= 1; end
f! (generic function with 1 method)

julia> a = [1]; f!(a); a
1-element Array{Int64,1}:
 2
```

Assignment can operate on multiple variables in parallel, taking values from an iterable:

```
julia> a, b = 4, 5
(4, 5)

julia> a, b = 1:3
1:3

julia> a, b
```

```
(1, 2)
```

Assignment can operate on multiple variables in series, and will return the value of the right-hand-most expression:

```
julia> a = [1]; b = [2]; c = [3]; a = b = c
1-element Array{Int64,1}:
 3

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

Assignment at out-of-bounds indices does not grow a collection. If the collection is a `Vector` it can instead be grown with `push!` or `append!`.

```
julia> a = [1, 1]; a[3] = 2
ERROR: BoundsError: attempt to access 2-element Array{Int64,1} at index [3]
[...]

julia> push!(a, 2, 3)
4-element Array{Int64,1}:
 1
 1
 2
 3
```

Assigning `[]` does not eliminate elements from a collection; instead use `filter!`.

```
julia> a = collect(1:3); a[a .<= 1] = []
ERROR: DimensionMismatch("tried to assign 0 elements to 1 destinations")
[...]

julia> filter!(x -> x > 1, a) # in-place & thus more efficient than a = a[a .>
2-element Array{Int64,1}:
 2
 3
```

`?:` — Keyword

```
a ? b : c
```

Short form for conditionals; read "if `a`, evaluate `b` otherwise evaluate `c`". Also known as the ternary operator.

This syntax is equivalent to `if a; b else c end`, but is often used to emphasize the value `b`-or-`c` which is being used as part of a larger expression, rather than the side effects that evaluating `b` or `c` may have.

See the manual section on control flow for more details.

Examples

```
julia> x = 1; y = 2;

julia> println(x > y ? "x is larger" : "y is larger")
y is larger
```

Standard Modules

`Main` — Module

```
Main
```

`Main` is the top-level module, and Julia starts with `Main` set as the current module. Variables defined at the prompt go in `Main`, and `varinfo` lists variables in `Main`.

```
julia> @__MODULE__
Main
```

`Core` — Module

```
Core
```

`Core` is the module that contains all identifiers considered "built in" to the language, i.e. part of the

core language and not libraries. Every module implicitly specifies `using Core`, since you can't do anything without those definitions.

`Base` — Module

```
Base
```

The base library of Julia. `Base` is a module that contains basic functionality (the contents of `base/`). All modules implicitly contain `using Base`, since this is needed in the vast majority of cases.

Base Submodules

`Base.Broadcast` — Module

```
Base.Broadcast
```

Module containing the broadcasting implementation.

`Base.Docs` — Module

```
Docs
```

The `Docs` module provides the `@doc` macro which can be used to set and retrieve documentation metadata for Julia objects.

Please see the manual section on documentation for more information.

`Base.Iterators` — Module

Methods for working with Iterators.

`Base.Libc` — Module

Interface to libc, the C standard library.

`Base.Meta` — Module

Convenience functions for metaprogramming.

`Base.StackTraces` — Module

Tools for collecting and manipulating stack traces. Mainly used for building errors.

`Base.Sys` — Module

Provide methods for retrieving information about hardware and the operating system.

`Base.Threads` — Module

Multithreading support.

`Base.GC` — Module

```
Base.GC
```

Module with garbage collection utilities.

All Objects

`Core.:===` — Function

```
===(x,y) -> Bool
```

```
≡(x,y) -> Bool
```

Determine whether `x` and `y` are identical, in the sense that no program could distinguish them. First the types of `x` and `y` are compared. If those are identical, mutable objects are compared by address in memory and immutable objects (such as numbers) are compared by contents at the bit level. This function is sometimes called "egal". It always returns a `Bool` value.

Examples

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

julia> a == b
true

julia> a === b
false

julia> a === a
true
```

`Core.isa` — Function

```
isa(x, type) -> Bool
```

Determine whether `x` is of the given `type`. Can also be used as an infix operator, e.g. `x isa type`.

Examples

```
julia> isa(1, Int)
true

julia> isa(1, Matrix)
false

julia> isa(1, Char)
false

julia> isa(1, Number)
true

julia> 1 isa Number
```

```
true
```

`Base.isequal` — Function

```
isequal(x, y)
```

Similar to `==`, except for the treatment of floating point numbers and of missing values. `isequal` treats all floating-point `NaN` values as equal to each other, treats `-0.0` as unequal to `0.0`, and `missing` as equal to `missing`. Always returns a `Bool` value.

Implementation

The default implementation of `isequal` calls `==`, so a type that does not involve floating-point values generally only needs to define `==`.

`isequal` is the comparison function used by hash tables (`Dict`). `isequal(x,y)` must imply that `hash(x) == hash(y)`.

This typically means that types for which a custom `==` or `isequal` method exists must implement a corresponding `hash` method (and vice versa). Collections typically implement `isequal` by calling `isequal` recursively on all contents.

Scalar types generally do not need to implement `isequal` separate from `==`, unless they represent floating-point numbers amenable to a more efficient implementation than that provided as a generic fallback (based on `isnan`, `signbit`, and `==`).

Examples

```
julia> isequal([1., NaN], [1., NaN])
true

julia> [1., NaN] == [1., NaN]
false

julia> 0.0 == -0.0
true

julia> isequal(0.0, -0.0)
false
```

```
isequal(x)
```

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

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

`Base.isless` — Function

```
isless(x, y)
```

Test whether `x` is less than `y`, according to a fixed total order. `isless` is not defined on all pairs of values `(x, y)`. However, if it is defined, it is expected to satisfy the following:

- If `isless(x, y)` is defined, then so is `isless(y, x)` and `isequal(x, y)`, and exactly one of those three yields `true`.
- The relation defined by `isless` is transitive, i.e., `isless(x, y) && isless(y, z)` implies `isless(x, z)`.

Values that are normally unordered, such as `NaN`, are ordered in an arbitrary but consistent fashion. `missing` values are ordered last.

This is the default comparison used by `sort`.

Implementation

Non-numeric types with a total order should implement this function. Numeric types only need to implement it if they have special values such as `NaN`. Types with a partial order should implement `<`.

Examples

```jldoctest julia> isless(1, 3) true

julia> isless("Red", "Blue") false ```

`Core.ifelse` — Function
```

```
ifelse(condition::Bool, x, y)
```

Return `x` if `condition` is `true`, otherwise return `y`. This differs from `?` or `if` in that it is an ordinary function, so all the arguments are evaluated first. In some cases, using `ifelse` instead of an `if` statement can eliminate the branch in generated code and provide higher performance in tight loops.

Examples

```
julia> ifelse(1 > 2, 1, 2)
2
```

`Core.typeassert` — Function

```
typeassert(x, type)
```

Throw a `TypeError` unless `x isa type`. The syntax `x::type` calls this function.

Examples

```
julia> typeassert(2.5, Int)
ERROR: TypeError: in typeassert, expected Int64, got a value of type Float64
Stacktrace:
[...]
```

`Core.typeof` — Function

```
typeof(x)
```

Get the concrete type of `x`.

Examples

```
julia> a = 1//2;

julia> typeof(a)
```

```
Rational{Int64}

julia> M = [1 2; 3.5 4];

julia> typeof(M)
Array{Float64,2}
```

`Core.tuple` — Function

```
tuple(xs...)
```

Construct a tuple of the given objects.

Examples

```
julia> tuple(1, 'a', pi)
(1, 'a', π)
```

`Base.ntuple` — Function

```
ntuple(f::Function, n::Integer)
```

Create a tuple of length `n`, computing each element as `f(i)`, where `i` is the index of the element.

Examples

```
julia> ntuple(i -> 2*i, 4)
(2, 4, 6, 8)
```

`Base.objectid` — Function

```
objectid(x)
```

Get a hash value for `x` based on object identity. `objectid(x)==objectid(y)` if `x === y`.

`Base.hash` — Function

```
hash(x[, h::UInt])
```

Compute an integer hash code such that `isequal(x,y)` implies `hash(x)==hash(y)`. The optional second argument `h` is a hash code to be mixed with the result.

New types should implement the 2-argument form, typically by calling the 2-argument `hash` method recursively in order to mix hashes of the contents with each other (and with `h`). Typically, any type that implements `hash` should also implement its own `==` (hence `isequal`) to guarantee the property mentioned above. Types supporting subtraction (operator `-`) should also implement `widen`, which is required to hash values inside heterogeneous arrays.

`Base.finalizer` — Function

```
finalizer(f, x)
```

Register a function `f(x)` to be called when there are no program-accessible references to `x`, and return `x`. The type of `x` must be a `mutable struct`, otherwise the behavior of this function is unpredictable.

`f` must not cause a task switch, which excludes most I/O operations such as `println`. Using the `@async` macro (to defer context switching to outside of the finalizer) or `ccall` to directly invoke IO functions in C may be helpful for debugging purposes.

Examples

```
finalizer(my_mutable_struct) do x
    @async println("Finalizing $x.")
end

finalizer(my_mutable_struct) do x
    ccall(:jl_safe_printf, Cvoid, (Cstring, Cstring), "Finalizing %s.", repr(x))
end
```

`Base.finalize` — Function

```
finalize(x)
```

Immediately run finalizers registered for object `x`.

`Base.copy` — Function

```
copy(x)
```

Create a shallow copy of `x`: the outer structure is copied, but not all internal values. For example, copying an array produces a new array with identically-same elements as the original.

`Base.deepcopy` — Function

```
deepcopy(x)
```

Create a deep copy of `x`: everything is copied recursively, resulting in a fully independent object. For example, deep-copying an array produces a new array whose elements are deep copies of the original elements. Calling `deepcopy` on an object should generally have the same effect as serializing and then deserializing it.

While it isn't normally necessary, user-defined types can override the default `deepcopy` behavior by defining a specialized version of the function `deepcopy_internal(x::T, dict::IdDict)` (which shouldn't otherwise be used), where `T` is the type to be specialized for, and `dict` keeps track of objects copied so far within the recursion. Within the definition, `deepcopy_internal` should be used in place of `deepcopy`, and the `dict` variable should be updated as appropriate before returning.

`Base.getproperty` — Function

```
getproperty(value, name::Symbol)
```

The syntax `a.b` calls `getproperty(a, :b)`.

Examples

```
julia> struct MyType
           x
       end

julia> function Base.getproperty(obj::MyType, sym::Symbol)
           if sym === :special
               return obj.x + 1
           else # fallback to getfield
               return getfield(obj, sym)
           end
       end

julia> obj = MyType(1);

julia> obj.special
2

julia> obj.x
1
```

See also `propertynames` and `setproperty!`.

`Base.setproperty!` — Function

```
setproperty!(value, name::Symbol, x)
```

The syntax `a.b = c` calls `setproperty!(a, :b, c)`.

See also `propertynames` and `getproperty`.

`Base.propertynames` — Function

```
propertynames(x, private=false)
```

Get a tuple or a vector of the properties (`x.property`) of an object `x`. This is typically the same as `fieldnames(typeof(x))`, but types that overload `getproperty` should generally overload `propertynames` as well to get the properties of an instance of the type.

`propertynames(x)` may return only "public" property names that are part of the documented

interface of `x`. If you want it to also return "private" fieldnames intended for internal use, pass `true` for the optional second argument. REPL tab completion on `x.` shows only the `private=false` properties.

`Base.hasproperty` — Function

```
hasproperty(x, s::Symbol)
```

Return a boolean indicating whether the object `x` has `s` as one of its own properties.

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

`Core.getfield` — Function

```
getfield(value, name::Symbol)
getfield(value, i::Int)
```

Extract a field from a composite `value` by name or position. See also `getproperty` and `fieldnames`.

Examples

```
julia> a = 1//2
1//2

julia> getfield(a, :num)
1

julia> a.num
1

julia> getfield(a, 1)
1
```

`Core.setfield!` — Function

```
setfield!(value, name::Symbol, x)
```

Assign `x` to a named field in `value` of composite type. The `value` must be mutable and `x` must be a subtype of `fieldtype(typeof(value), name)`. See also `setproperty!`.

Examples

```
julia> mutable struct MyMutableStruct
           field::Int
       end

julia> a = MyMutableStruct(1);

julia> setfield!(a, :field, 2);

julia> getfield(a, :field)
2

julia> a = 1//2
1//2

julia> setfield!(a, :num, 3);
ERROR: setfield! immutable struct of type Rational cannot be changed
```

`Core.isdefined` — Function

```
isdefined(m::Module, s::Symbol)
isdefined(object, s::Symbol)
isdefined(object, index::Int)
```

Tests whether a global variable or object field is defined. The arguments can be a module and a symbol or a composite object and field name (as a symbol) or index.

To test whether an array element is defined, use `isassigned` instead.

See also `@isdefined`.

Examples

```
julia> isdefined(Base, :sum)
true

julia> isdefined(Base, :NonExistentMethod)
false

julia> a = 1//2;

julia> isdefined(a, 2)
true

julia> isdefined(a, 3)
false

julia> isdefined(a, :num)
true

julia> isdefined(a, :numerator)
false
```

`Base.@isdefined` — Macro

```
@isdefined s -> Bool
```

Tests whether variable `s` is defined in the current scope.

See also `isdefined`.

Examples

```
julia> function f()
           println(@isdefined x)
           x = 3
           println(@isdefined x)
       end
f (generic function with 1 method)

julia> f()
false
true
```

`Base.convert` — Function

```
convert(T, x)
```

Convert `x` to a value of type `T`.

If `T` is an `Integer` type, an `InexactError` will be raised if `x` is not representable by `T`, for example if `x` is not integer-valued, or is outside the range supported by `T`.

Examples

```
julia> convert(Int, 3.0)
3

julia> convert(Int, 3.5)
ERROR: InexactError: Int64(3.5)
Stacktrace:
[...]
```

If `T` is a `AbstractFloat` or `Rational` type, then it will return the closest value to `x` representable by `T`.

```
julia> x = 1/3
0.3333333333333333

julia> convert(Float32, x)
0.33333334f0

julia> convert(Rational{Int32}, x)
1//3

julia> convert(Rational{Int64}, x)
6004799503160661//18014398509481984
```

If `T` is a collection type and `x` a collection, the result of `convert(T, x)` may alias all or part of `x`.

```
julia> x = Int[1, 2, 3];

julia> y = convert(Vector{Int}, x);

julia> y === x
```

```
true
```

Base.promote — Function

```
promote(xs...)
```

Convert all arguments to a common type, and return them all (as a tuple). If no arguments can be converted, an error is raised.

Examples

```
julia> promote(Int8(1), Float16(4.5), Float32(4.1))
(1.0f0, 4.5f0, 4.1f0)
```

Base.oftype — Function

```
oftype(x, y)
```

Convert `y` to the type of `x` (`convert(typeof(x), y)`).

Examples

```
julia> x = 4;

julia> y = 3.;

julia> oftype(x, y)
3

julia> oftype(y, x)
4.0
```

Base.widen — Function

```
widen(x)
```

If `x` is a type, return a "larger" type, defined so that arithmetic operations `+` and `-` are guaranteed not to overflow nor lose precision for any combination of values that type `x` can hold.

For fixed-size integer types less than 128 bits, `widen` will return a type with twice the number of bits.

If `x` is a value, it is converted to `widen(typeof(x))`.

Examples

```
julia> widen(Int32)
Int64

julia> widen(1.5f0)
1.5
```

`Base.identity` — Function

```
identity(x)
```

The identity function. Returns its argument.

Examples

```
julia> identity("Well, what did you expect?")
"Well, what did you expect?"
```

Properties of Types

Type relations

`Base.supertype` — Function

```
supertype(T::DataType)
```

Return the supertype of DataType `T`.

Examples

```
julia> supertype(Int32)
Signed
```

Core.Type — Type

```
Core.Type{T}
```

`Core.Type` is an abstract type which has all type objects as its instances. The only instance of the singleton type `Core.Type{T}` is the object `T`.

Examples

```
julia> isa(Type{Float64}, Type)
true

julia> isa(Float64, Type)
true

julia> isa(Real, Type{Float64})
false

julia> isa(Real, Type{Real})
true
```

Core.DataType — Type

```
DataType <: Type{T}
```

`DataType` represents explicitly declared types that have names, explicitly declared supertypes, and, optionally, parameters. Every concrete value in the system is an instance of some `DataType`.

Examples

```
julia> typeof(Real)
DataType
```

```
julia> typeof(Int)
DataType

julia> struct Point
           x::Int
           y
       end

julia> typeof(Point)
DataType
```

Core.:<: — Function

```
<:(T1, T2)
```

Subtype operator: returns `true` if and only if all values of type `T1` are also of type `T2`.

Examples

```
julia> Float64 <: AbstractFloat
true

julia> Vector{Int} <: AbstractArray
true

julia> Matrix{Float64} <: Matrix{AbstractFloat}
false
```

Base.:>: — Function

```
>:(T1, T2)
```

Supertype operator, equivalent to `T2 <: T1`.

Base.typejoin — Function

```
typejoin(T, S)
```

Return the closest common ancestor of `T` and `S`, i.e. the narrowest type from which they both inherit.

`Base.typeintersect` — Function

```
typeintersect(T, S)
```

Compute a type that contains the intersection of `T` and `S`. Usually this will be the smallest such type or one close to it.

`Base.promote_type` — Function

```
promote_type(type1, type2)
```

Promotion refers to converting values of mixed types to a single common type. `promote_type` represents the default promotion behavior in Julia when operators (usually mathematical) are given arguments of differing types. `promote_type` generally tries to return a type which can at least approximate most values of either input type without excessively widening. Some loss is tolerated; for example, `promote_type(Int64, Float64)` returns `Float64` even though strictly, not all `Int64` values can be represented exactly as `Float64` values.

```
julia> promote_type(Int64, Float64)
Float64

julia> promote_type(Int32, Int64)
Int64

julia> promote_type(Float32, BigInt)
BigFloat

julia> promote_type(Int16, Float16)
Float16

julia> promote_type(Int64, Float16)
Float16
```

```
julia> promote_type(Int8, UInt16)
UInt16
```

`Base.promote_rule` — Function

```
promote_rule(type1, type2)
```

Specifies what type should be used by `promote` when given values of types `type1` and `type2`. This function should not be called directly, but should have definitions added to it for new types as appropriate.

`Base.isdispatchtuple` — Function

```
isdispatchtuple(T)
```

Determine whether type `T` is a tuple "leaf type", meaning it could appear as a type signature in dispatch and has no subtypes (or supertypes) which could appear in a call.

Declared structure

`Base.ismutable` — Function

```
ismutable(v) -> Bool
```

Return `true` iff value `v` is mutable. See Mutable Composite Types for a discussion of immutability. Note that this function works on values, so if you give it a type, it will tell you that a value of `DataType` is mutable.

Examples

```
julia> ismutable(1)
false

julia> ismutable([1,2])
```

```
true
```

> **Julia 1.5**
>
> This function requires at least Julia 1.5.

`Base.isimmutable` — Function

```
isimmutable(v) -> Bool
```

> **Warning**
>
> Consider using `!ismutable(v)` instead, as `isimmutable(v)` will be replaced by `!ismutable(v)` in a future release. (Since Julia 1.5)

Return `true` iff value `v` is immutable. See Mutable Composite Types for a discussion of immutability. Note that this function works on values, so if you give it a type, it will tell you that a value of `DataType` is mutable.

Examples

```
julia> isimmutable(1)
true

julia> isimmutable([1,2])
false
```

`Base.isabstracttype` — Function

```
isabstracttype(T)
```

Determine whether type `T` was declared as an abstract type (i.e. using the `abstract` keyword).

Examples

```
julia> isabstracttype(AbstractArray)
true

julia> isabstracttype(Vector)
false
```

Base.isprimitivetype — Function

```
isprimitivetype(T) -> Bool
```

Determine whether type `T` was declared as a primitive type (i.e. using the `primitive` keyword).

Base.issingletontype — Function

```
Base.issingletontype(T)
```

Determine whether type `T` has exactly one possible instance; for example, a struct type with no fields.

Base.isstructtype — Function

```
isstructtype(T) -> Bool
```

Determine whether type `T` was declared as a struct type (i.e. using the `struct` or `mutable struct` keyword).

Base.nameof — Method

```
nameof(t::DataType) -> Symbol
```

Get the name of a (potentially `UnionAll`-wrapped) `DataType` (without its parent module) as a symbol.

Examples

```
julia> module Foo
           struct S{T}
           end
       end
Foo

julia> nameof(Foo.S{T} where T)
:S
```

`Base.fieldnames` — Function

```
fieldnames(x::DataType)
```

Get a tuple with the names of the fields of a `DataType`.

Examples

```
julia> fieldnames(Rational)
(:num, :den)
```

`Base.fieldname` — Function

```
fieldname(x::DataType, i::Integer)
```

Get the name of field `i` of a `DataType`.

Examples

```
julia> fieldname(Rational, 1)
:num

julia> fieldname(Rational, 2)
:den
```

`Base.hasfield` — Function

```
hasfield(T::Type, name::Symbol)
```

Return a boolean indicating whether `T` has `name` as one of its own fields.

> Julia 1.2
>
> This function requires at least Julia 1.2.

Memory layout

`Base.sizeof` — Method

```
sizeof(T::DataType)
sizeof(obj)
```

Size, in bytes, of the canonical binary representation of the given `DataType T`, if any. Size, in bytes, of object `obj` if it is not `DataType`.

Examples

```
julia> sizeof(Float32)
4

julia> sizeof(ComplexF64)
16

julia> sizeof(1.0)
8

julia> sizeof([1.0:10.0;])
80
```

If `DataType T` does not have a specific size, an error is thrown.

```
julia> sizeof(AbstractArray)
```

```
ERROR: Abstract type AbstractArray does not have a definite size.
Stacktrace:
[...]
```

`Base.isconcretetype` — Function

```
isconcretetype(T)
```

Determine whether type `T` is a concrete type, meaning it could have direct instances (values `x` such that `typeof(x) === T`).

Examples

```
julia> isconcretetype(Complex)
false

julia> isconcretetype(Complex{Float32})
true

julia> isconcretetype(Vector{Complex})
true

julia> isconcretetype(Vector{Complex{Float32}})
true

julia> isconcretetype(Union{})
false

julia> isconcretetype(Union{Int,String})
false
```

`Base.isbits` — Function

```
isbits(x)
```

Return `true` if `x` is an instance of an `isbitstype` type.

`Base.isbitstype` — Function

```
isbitstype(T)
```

Return `true` if type `T` is a "plain data" type, meaning it is immutable and contains no references to other values, only `primitive` types and other `isbitstype` types. Typical examples are numeric types such as `UInt8`, `Float64`, and `Complex{Float64}`. This category of types is significant since they are valid as type parameters, may not track `isdefined` / `isassigned` status, and have a defined layout that is compatible with C.

Examples

```
julia> isbitstype(Complex{Float64})
true

julia> isbitstype(Complex)
false
```

`Core.fieldtype` — Function

```
fieldtype(T, name::Symbol | index::Int)
```

Determine the declared type of a field (specified by name or index) in a composite DataType `T`.

Examples

```
julia> struct Foo
           x::Int64
           y::String
       end

julia> fieldtype(Foo, :x)
Int64

julia> fieldtype(Foo, 2)
String
```

Base.fieldtypes — Function

```
fieldtypes(T::Type)
```

The declared types of all fields in a composite DataType `T` as a tuple.

> **Julia 1.1**
>
> This function requires at least Julia 1.1.

Examples

```
julia> struct Foo
           x::Int64
           y::String
       end

julia> fieldtypes(Foo)
(Int64, String)
```

Base.fieldcount — Function

```
fieldcount(t::Type)
```

Get the number of fields that an instance of the given type would have. An error is thrown if the type is too abstract to determine this.

Base.fieldoffset — Function

```
fieldoffset(type, i)
```

The byte offset of field `i` of a type relative to the data start. For example, we could use it in the following manner to summarize information about a struct:

```
julia> structinfo(T) = [(fieldoffset(T,i), fieldname(T,i), fieldtype(T,i)) for
```

```
julia> structinfo(Base.Filesystem.StatStruct)
12-element Array{Tuple{UInt64,Symbol,DataType},1}:
 (0x0000000000000000, :device, UInt64)
 (0x0000000000000008, :inode, UInt64)
 (0x0000000000000010, :mode, UInt64)
 (0x0000000000000018, :nlink, Int64)
 (0x0000000000000020, :uid, UInt64)
 (0x0000000000000028, :gid, UInt64)
 (0x0000000000000030, :rdev, UInt64)
 (0x0000000000000038, :size, Int64)
 (0x0000000000000040, :blksize, Int64)
 (0x0000000000000048, :blocks, Int64)
 (0x0000000000000050, :mtime, Float64)
 (0x0000000000000058, :ctime, Float64)
```

`Base.datatype_alignment` — Function

```
Base.datatype_alignment(dt::DataType) -> Int
```

Memory allocation minimum alignment for instances of this type. Can be called on any `isconcretetype`.

`Base.datatype_haspadding` — Function

```
Base.datatype_haspadding(dt::DataType) -> Bool
```

Return whether the fields of instances of this type are packed in memory, with no intervening padding bytes. Can be called on any `isconcretetype`.

`Base.datatype_pointerfree` — Function

```
Base.datatype_pointerfree(dt::DataType) -> Bool
```

Return whether instances of this type can contain references to gc-managed memory. Can be called on any `isconcretetype`.

Special values

`Base.typemin` — Function

```
typemin(T)
```

The lowest value representable by the given (real) numeric DataType `T`.

Examples

```
julia> typemin(Float16)
-Inf16

julia> typemin(Float32)
-Inf32
```

`Base.typemax` — Function

```
typemax(T)
```

The highest value representable by the given (real) numeric `DataType`.

Examples

```
julia> typemax(Int8)
127

julia> typemax(UInt32)
0xffffffff
```

`Base.floatmin` — Function

```
floatmin(T = Float64)
```

Return the smallest positive normal number representable by the floating-point type `T`.

Examples

```
julia> floatmin(Float16)
Float16(6.104e-5)

julia> floatmin(Float32)
1.1754944f-38

julia> floatmin()
2.2250738585072014e-308
```

`Base.floatmax` — Function

```
floatmax(T = Float64)
```

Return the largest finite number representable by the floating-point type `T`.

Examples

```
julia> floatmax(Float16)
Float16(6.55e4)

julia> floatmax(Float32)
3.4028235f38

julia> floatmax()
1.7976931348623157e308
```

`Base.maxintfloat` — Function

```
maxintfloat(T=Float64)
```

The largest consecutive integer-valued floating-point number that is exactly represented in the given floating-point type `T` (which defaults to `Float64`).

That is, `maxintfloat` returns the smallest positive integer-valued floating-point number `n` such that `n+1` is *not* exactly representable in the type `T`.

When an `Integer`-type value is needed, use `Integer(maxintfloat(T))`.

```
maxintfloat(T, S)
```

The largest consecutive integer representable in the given floating-point type `T` that also does not exceed the maximum integer representable by the integer type `S`. Equivalently, it is the minimum of `maxintfloat(T)` and `typemax(S)`.

`Base.eps` — Method

```
eps(::Type{T}) where T<:AbstractFloat
eps()
```

Return the *machine epsilon* of the floating point type `T` (`T = Float64` by default). This is defined as the gap between 1 and the next largest value representable by `typeof(one(T))`, and is equivalent to `eps(one(T))`. (Since `eps(T)` is a bound on the *relative error* of `T`, it is a "dimensionless" quantity like `one`.)

Examples

```
julia> eps()
2.220446049250313e-16

julia> eps(Float32)
1.1920929f-7

julia> 1.0 + eps()
1.0000000000000002

julia> 1.0 + eps()/2
1.0
```

`Base.eps` — Method

```
eps(x::AbstractFloat)
```

Return the *unit in last place* (ulp) of `x`. This is the distance between consecutive representable

floating point values at `x`. In most cases, if the distance on either side of `x` is different, then the larger of the two is taken, that is

```
eps(x) == max(x-prevfloat(x), nextfloat(x)-x)
```

The exceptions to this rule are the smallest and largest finite values (e.g. `nextfloat(-Inf)` and `prevfloat(Inf)` for `Float64`), which round to the smaller of the values.

The rationale for this behavior is that `eps` bounds the floating point rounding error. Under the default `RoundNearest` rounding mode, if y is a real number and x is the nearest floating point number to y, then

$$|y - x| \leq \operatorname{eps}(x)/2.$$

Examples

```
julia> eps(1.0)
2.220446049250313e-16

julia> eps(prevfloat(2.0))
2.220446049250313e-16

julia> eps(2.0)
4.440892098500626e-16

julia> x = prevfloat(Inf)      # largest finite Float64
1.7976931348623157e308

julia> x + eps(x)/2            # rounds up
Inf

julia> x + prevfloat(eps(x)/2) # rounds down
1.7976931348623157e308
```

`Base.instances` — Function

```
instances(T::Type)
```

Return a collection of all instances of the given type, if applicable. Mostly used for enumerated types (see `@enum`).

Example

```
julia> @enum Color red blue green

julia> instances(Color)
(red, blue, green)
```

Special Types

`Core.Any` — Type

```
Any::DataType
```

`Any` is the union of all types. It has the defining property `isa(x, Any) == true` for any `x`. `Any` therefore describes the entire universe of possible values. For example `Integer` is a subset of `Any` that includes `Int`, `Int8`, and other integer types.

`Core.Union` — Type

```
Union{Types...}
```

A type union is an abstract type which includes all instances of any of its argument types. The empty union `Union{}` is the bottom type of Julia.

Examples

```
julia> IntOrString = Union{Int,AbstractString}
Union{Int64, AbstractString}

julia> 1 :: IntOrString
1

julia> "Hello!" :: IntOrString
"Hello!"

julia> 1.0 :: IntOrString
ERROR: TypeError: in typeassert, expected Union{Int64, AbstractString}, got a v
```

`Union{}` — Keyword

```
Union{}
```

`Union{}`, the empty `Union` of types, is the type that has no values. That is, it has the defining property `isa(x, Union{}) == false` for any `x`. `Base.Bottom` is defined as its alias and the type of `Union{}` is `Core.TypeofBottom`.

Examples

```
julia> isa(nothing, Union{})
false
```

`Core.UnionAll` — Type

```
UnionAll
```

A union of types over all values of a type parameter. `UnionAll` is used to describe parametric types where the values of some parameters are not known.

Examples

```
julia> typeof(Vector)
UnionAll

julia> typeof(Vector{Int})
DataType
```

`Core.Tuple` — Type

```
Tuple{Types...}
```

Tuples are an abstraction of the arguments of a function – without the function itself. The salient aspects of a function's arguments are their order and their types. Therefore a tuple type is similar to a parameterized immutable type where each parameter is the type of one field. Tuple types may have any number of parameters.

Tuple types are covariant in their parameters: `Tuple{Int}` is a subtype of `Tuple{Any}`. Therefore `Tuple{Any}` is considered an abstract type, and tuple types are only concrete if their parameters are. Tuples do not have field names; fields are only accessed by index.

See the manual section on Tuple Types.

`Core.NamedTuple` — Type

```
NamedTuple
```

`NamedTuples` are, as their name suggests, named `Tuples`. That is, they're a tuple-like collection of values, where each entry has a unique name, represented as a `Symbol`. Like `Tuples`, `NamedTuples` are immutable; neither the names nor the values can be modified in place after construction.

Accessing the value associated with a name in a named tuple can be done using field access syntax, e.g. `x.a`, or using `getindex`, e.g. `x[:a]`. A tuple of the names can be obtained using `keys`, and a tuple of the values can be obtained using `values`.

> **Note**
>
> Iteration over `NamedTuple`s produces the *values* without the names. (See example below.) To iterate over the name-value pairs, use the `pairs` function.

The `@NamedTuple` macro can be used for conveniently declaring `NamedTuple` types.

Examples

```
julia> x = (a=1, b=2)
(a = 1, b = 2)

julia> x.a
1

julia> x[:a]
1

julia> keys(x)
(:a, :b)

julia> values(x)
(1, 2)

julia> collect(x)
2-element Array{Int64,1}:
 1
 2

julia> collect(pairs(x))
2-element Array{Pair{Symbol,Int64},1}:
 :a => 1
 :b => 2
```

In a similar fashion as to how one can define keyword arguments programmatically, a named tuple can be created by giving a pair `name::Symbol => value` or splatting an iterator yielding such pairs after a semicolon inside a tuple literal:

```
julia> (; :a => 1)
(a = 1,)

julia> keys = (:a, :b, :c); values = (1, 2, 3);
```

```
julia> (; zip(keys, values)...)
(a = 1, b = 2, c = 3)
```

As in keyword arguments, identifiers and dot expressions imply names:

```
julia> x = 0
0

julia> t = (; x)
(x = 0,)

julia> (; t.x)
(x = 0,)
```

> **Julia 1.5**
>
> Implicit names from identifiers and dot expressions are available as of Julia 1.5.

`Base.@NamedTuple` — Macro

```
@NamedTuple{key1::Type1, key2::Type2, ...}
@NamedTuple begin key1::Type1; key2::Type2; ...; end
```

This macro gives a more convenient syntax for declaring `NamedTuple` types. It returns a `NamedTuple` type with the given keys and types, equivalent to `NamedTuple{(:key1, :key2, ...), Tuple{Type1,Type2,...}}`. If the `::Type` declaration is omitted, it is taken to be `Any`. The `begin ... end` form allows the declarations to be split across multiple lines (similar to a `struct` declaration), but is otherwise equivalent.

For example, the tuple `(a=3.1, b="hello")` has a type `NamedTuple{(:a, :b),Tuple{Float64,String}}`, which can also be declared via `@NamedTuple` as:

```
julia> @NamedTuple{a::Float64, b::String}
NamedTuple{(:a, :b),Tuple{Float64,String}}

julia> @NamedTuple begin
           a::Float64
           b::String
```

```
    end
NamedTuple{(:a, :b),Tuple{Float64,String}}
```

> **Julia 1.5**
>
> This macro is available as of Julia 1.5.

`Base.Val` — Type

```
Val(c)
```

Return `Val{c}()`, which contains no run-time data. Types like this can be used to pass the information between functions through the value `c`, which must be an `isbits` value. The intent of this construct is to be able to dispatch on constants directly (at compile time) without having to test the value of the constant at run time.

Examples

```
julia> f(::Val{true}) = "Good"
f (generic function with 1 method)

julia> f(::Val{false}) = "Bad"
f (generic function with 2 methods)

julia> f(Val(true))
"Good"
```

`Core.Vararg` — Type

```
Vararg{T,N}
```

The last parameter of a tuple type `Tuple` can be the special type `Vararg`, which denotes any number of trailing elements. The type `Vararg{T,N}` corresponds to exactly `N` elements of type `T`. `Vararg{T}` corresponds to zero or more elements of type `T`. `Vararg` tuple types are used to represent the arguments accepted by varargs methods (see the section on Varargs Functions in the manual.)

Examples

```
julia> mytupletype = Tuple{AbstractString,Vararg{Int}}
Tuple{AbstractString,Vararg{Int64,N} where N}

julia> isa(("1",), mytupletype)
true

julia> isa(("1",1), mytupletype)
true

julia> isa(("1",1,2), mytupletype)
true

julia> isa(("1",1,2,3.0), mytupletype)
false
```

`Core.Nothing` — Type

```
Nothing
```

A type with no fields that is the type of `nothing`.

`Base.isnothing` — Function

```
isnothing(x)
```

Return `true` if `x === nothing`, and return `false` if not.

> **Julia 1.1**
>
> This function requires at least Julia 1.1.

`Base.Some` — Type

```
Some{T}
```

A wrapper type used in `Union{Some{T}, Nothing}` to distinguish between the absence of a value (`nothing`) and the presence of a `nothing` value (i.e. `Some(nothing)`).

Use `something` to access the value wrapped by a `Some` object.

`Base.something` — Function

```
something(x, y...)
```

Return the first value in the arguments which is not equal to `nothing`, if any. Otherwise throw an error. Arguments of type `Some` are unwrapped.

See also `coalesce`.

Examples

```
julia> something(nothing, 1)
1

julia> something(Some(1), nothing)
1

julia> something(missing, nothing)
missing

julia> something(nothing, nothing)
ERROR: ArgumentError: No value arguments present
```

`Base.Enums.Enum` — Type

```
Enum{T<:Integer}
```

The abstract supertype of all enumerated types defined with `@enum`.

`Base.Enums.@enum` — Macro

```
@enum EnumName[::BaseType] value1[=x] value2[=y]
```

Create an `Enum{BaseType}` subtype with name `EnumName` and enum member values of `value1` and `value2` with optional assigned values of `x` and `y`, respectively. `EnumName` can be used just like other types and enum member values as regular values, such as

Examples

```
julia> @enum Fruit apple=1 orange=2 kiwi=3

julia> f(x::Fruit) = "I'm a Fruit with value: $(Int(x))"
f (generic function with 1 method)

julia> f(apple)
"I'm a Fruit with value: 1"

julia> Fruit(1)
apple::Fruit = 1
```

Values can also be specified inside a `begin` block, e.g.

```
@enum EnumName begin
    value1
    value2
end
```

`BaseType`, which defaults to `Int32`, must be a primitive subtype of `Integer`. Member values can be converted between the enum type and `BaseType`. `read` and `write` perform these conversions automatically.

To list all the instances of an enum use `instances`, e.g.

```
julia> instances(Fruit)
(apple, orange, kiwi)
```

`Core.Expr` — Type

```
Expr(head::Symbol, args...)
```

A type representing compound expressions in parsed julia code (ASTs). Each expression consists of a `head` `Symbol` identifying which kind of expression it is (e.g. a call, for loop, conditional statement, etc.), and subexpressions (e.g. the arguments of a call). The subexpressions are stored in a `Vector{Any}` field called `args`.

See the manual chapter on Metaprogramming and the developer documentation Julia ASTs.

Examples

```
julia> Expr(:call, :+, 1, 2)
:(1 + 2)

julia> dump(:(a ? b : c))
Expr
  head: Symbol if
  args: Array{Any}((3,))
    1: Symbol a
    2: Symbol b
    3: Symbol c
```

`Core.Symbol` — Type

```
Symbol
```

The type of object used to represent identifiers in parsed julia code (ASTs). Also often used as a name or label to identify an entity (e.g. as a dictionary key). `Symbol`s can be entered using the `:` quote operator:

```
julia> :name
:name

julia> typeof(:name)
Symbol

julia> x = 42
42
```

```
julia> eval(:x)
42
```

`Symbol`s can also be constructed from strings or other values by calling the constructor `Symbol(x...)`.

`Symbol`s are immutable and should be compared using `===`. The implementation re-uses the same object for all `Symbol`s with the same name, so comparison tends to be efficient (it can just compare pointers).

Unlike strings, `Symbol`s are "atomic" or "scalar" entities that do not support iteration over characters.

`Core.Symbol` — Method

```
Symbol(x...) -> Symbol
```

Create a `Symbol` by concatenating the string representations of the arguments together.

Examples

```
julia> Symbol("my", "name")
:myname

julia> Symbol("day", 4)
:day4
```

`Core.Module` — Type

```
Module
```

A `Module` is a separate global variable workspace. See `module` and the manual section about modules for details.

Generic Functions

`Core.Function` — Type

```
Function
```

Abstract type of all functions.

Examples

```
julia> isa(+, Function)
true

julia> typeof(sin)
typeof(sin)

julia> ans <: Function
true
```

`Base.hasmethod` — Function

```
hasmethod(f, t::Type{<:Tuple}[, kwnames]; world=typemax(UInt)) -> Bool
```

Determine whether the given generic function has a method matching the given `Tuple` of argument types with the upper bound of world age given by `world`.

If a tuple of keyword argument names `kwnames` is provided, this also checks whether the method of `f` matching `t` has the given keyword argument names. If the matching method accepts a variable number of keyword arguments, e.g. with `kwargs...`, any names given in `kwnames` are considered valid. Otherwise the provided names must be a subset of the method's keyword arguments.

See also `applicable`.

> **Julia 1.2**
>
> Providing keyword argument names requires Julia 1.2 or later.

Examples

```
julia> hasmethod(length, Tuple{Array})
true

julia> hasmethod(sum, Tuple{Function, Array}, (:dims,))
true

julia> hasmethod(sum, Tuple{Function, Array}, (:apples, :bananas))
false

julia> g(; xs...) = 4;

julia> hasmethod(g, Tuple{}, (:a, :b, :c, :d))  # g accepts arbitrary kwargs
true
```

`Core.applicable` — Function

```
applicable(f, args...) -> Bool
```

Determine whether the given generic function has a method applicable to the given arguments.

See also `hasmethod`.

Examples

```
julia> function f(x, y)
           x + y
       end;

julia> applicable(f, 1)
false

julia> applicable(f, 1, 2)
true
```

`Core.invoke` — Function

```
invoke(f, argtypes::Type, args...; kwargs...)
```

Invoke a method for the given generic function `f` matching the specified types `argtypes` on the specified arguments `args` and passing the keyword arguments `kwargs`. The arguments `args` must conform with the specified types in `argtypes`, i.e. conversion is not automatically performed. This method allows invoking a method other than the most specific matching method, which is useful when the behavior of a more general definition is explicitly needed (often as part of the implementation of a more specific method of the same function).

Be careful when using `invoke` for functions that you don't write. What definition is used for given `argtypes` is an implementation detail unless the function is explicitly states that calling with certain `argtypes` is a part of public API. For example, the change between `f1` and `f2` in the example below is usually considered compatible because the change is invisible by the caller with a normal (non-`invoke`) call. However, the change is visible if you use `invoke`.

Examples

```
julia> f(x::Real) = x^2;

julia> f(x::Integer) = 1 + invoke(f, Tuple{Real}, x);

julia> f(2)
5

julia> f1(::Integer) = Integer
       f1(::Real) = Real;

julia> f2(x::Real) = _f2(x)
       _f2(::Integer) = Integer
       _f2(_) = Real;

julia> f1(1)
Integer

julia> f2(1)
Integer

julia> invoke(f1, Tuple{Real}, 1)
Real

julia> invoke(f2, Tuple{Real}, 1)
Integer
```

`Base.invokelatest` — Function

```
invokelatest(f, args...; kwargs...)
```

Calls `f(args...; kwargs...)`, but guarantees that the most recent method of `f` will be executed. This is useful in specialized circumstances, e.g. long-running event loops or callback functions that may call obsolete versions of a function `f`. (The drawback is that `invokelatest` is somewhat slower than calling `f` directly, and the type of the result cannot be inferred by the compiler.)

`new` — Keyword

```
new
```

Special function available to inner constructors which created a new object of the type. See the manual section on Inner Constructor Methods for more information.

`Base.:|>` — Function

```
|>(x, f)
```

Applies a function to the preceding argument. This allows for easy function chaining.

Examples

```
julia> [1:5;] |> x->x.^2 |> sum |> inv
0.01818181818181818
```

`Base.:∘` — Function

```
f ∘ g
```

Compose functions: i.e. `(f ∘ g)(args...)` means `f(g(args...))`. The `∘` symbol can be entered in the Julia REPL (and most editors, appropriately configured) by typing `\circ<tab>`.

Function composition also works in prefix form: `∘(f, g)` is the same as `f ∘ g`. The prefix form supports composition of multiple functions: `∘(f, g, h) = f ∘ g ∘ h` and splatting `∘(fs...)` for composing an iterable collection of functions.

> **Julia 1.4**
>
> Multiple function composition requires at least Julia 1.4.

> **Julia 1.5**
>
> Composition of one function ∘(f) requires at least Julia 1.5.

Examples

```
julia> map(uppercase∘first, ["apple", "banana", "carrot"])
3-element Array{Char,1}:
 'A': ASCII/Unicode U+0041 (category Lu: Letter, uppercase)
 'B': ASCII/Unicode U+0042 (category Lu: Letter, uppercase)
 'C': ASCII/Unicode U+0043 (category Lu: Letter, uppercase)

julia> fs = [
           x -> 2x
           x -> x/2
           x -> x-1
           x -> x+1
       ];

julia> ∘(fs...)(3)
3.0
```

Syntax

`Core.eval` — Function

```
Core.eval(m::Module, expr)
```

Evaluate an expression in the given module and return the result.

`Base.MainInclude.eval` — Function

```
eval(expr)
```

Evaluate an expression in the global scope of the containing module. Every `Module` (except those defined with `baremodule`) has its own 1-argument definition of `eval`, which evaluates expressions in that module.

`Base.@eval` — Macro

```
@eval [mod,] ex
```

Evaluate an expression with values interpolated into it using `eval`. If two arguments are provided, the first is the module to evaluate in.

`Base.evalfile` — Function

```
evalfile(path::AbstractString, args::Vector{String}=String[])
```

Load the file using `include`, evaluate all expressions, and return the value of the last one.

`Base.esc` — Function

```
esc(e)
```

Only valid in the context of an `Expr` returned from a macro. Prevents the macro hygiene pass from turning embedded variables into gensym variables. See the Macros section of the Metaprogramming chapter of the manual for more details and examples.

`Base.@inbounds` — Macro

```
@inbounds(blk)
```

Eliminates array bounds checking within expressions.

In the example below the in-range check for referencing element `i` of array `A` is skipped to improve performance.

```
function sum(A::AbstractArray)
    r = zero(eltype(A))
    for i = 1:length(A)
        @inbounds r += A[i]
    end
    return r
end
```

Warning

Using `@inbounds` may return incorrect results/crashes/corruption for out-of-bounds indices. The user is responsible for checking it manually. Only use `@inbounds` when it is certain from the information locally available that all accesses are in bounds.

`Base.@boundscheck` — Macro

```
@boundscheck(blk)
```

Annotates the expression `blk` as a bounds checking block, allowing it to be elided by `@inbounds`.

Note

The function in which `@boundscheck` is written must be inlined into its caller in order for `@inbounds` to have effect.

Examples

```
julia> @inline function g(A, i)
           @boundscheck checkbounds(A, i)
```

```
            return "accessing ($A)[$i]"
        end;

julia> f1() = return g(1:2, -1);

julia> f2() = @inbounds return g(1:2, -1);

julia> f1()
ERROR: BoundsError: attempt to access 2-element UnitRange{Int64} at index [-1]
Stacktrace:
 [1] throw_boundserror(::UnitRange{Int64}, ::Tuple{Int64}) at ./abstractarray.j
 [2] checkbounds at ./abstractarray.jl:420 [inlined]
 [3] g at ./none:2 [inlined]
 [4] f1() at ./none:1
 [5] top-level scope

julia> f2()
"accessing (1:2)[-1]"
```

Warning

The `@boundscheck` annotation allows you, as a library writer, to opt-in to allowing *other code* to remove your bounds checks with `@inbounds`. As noted there, the caller must verify—using information they can access—that their accesses are valid before using `@inbounds`. For indexing into your `AbstractArray` subclasses, for example, this involves checking the indices against its `size`. Therefore, `@boundscheck` annotations should only be added to a `getindex` or `setindex!` implementation after you are certain its behavior is correct.

`Base.@propagate_inbounds` — Macro

```
@propagate_inbounds
```

Tells the compiler to inline a function while retaining the caller's inbounds context.

`Base.@inline` — Macro

```
@inline
```

Give a hint to the compiler that this function is worth inlining.

Small functions typically do not need the `@inline` annotation, as the compiler does it automatically. By using `@inline` on bigger functions, an extra nudge can be given to the compiler to inline it. This is shown in the following example:

```
@inline function bigfunction(x)
    #=
        Function Definition
    =#
end
```

`Base.@noinline` — Macro

```
@noinline
```

Give a hint to the compiler that it should not inline a function.

Small functions are typically inlined automatically. By using `@noinline` on small functions, auto-inlining can be prevented. This is shown in the following example:

```
@noinline function smallfunction(x)
    #=
        Function Definition
    =#
end

If the function is trivial (for example returning a constant) it might get inl:
```

`Base.@nospecialize` — Macro

```
@nospecialize
```

Applied to a function argument name, hints to the compiler that the method should not be specialized for different types of that argument, but instead to use precisely the declared type for each argument. This is only a hint for avoiding excess code generation. Can be applied to an argument within a formal argument list, or in the function body. When applied to an argument, the

macro must wrap the entire argument expression. When used in a function body, the macro must occur in statement position and before any code.

When used without arguments, it applies to all arguments of the parent scope. In local scope, this means all arguments of the containing function. In global (top-level) scope, this means all methods subsequently defined in the current module.

Specialization can reset back to the default by using `@specialize`.

```
function example_function(@nospecialize x)
    ...
end

function example_function(@nospecialize(x = 1), y)
    ...
end

function example_function(x, y, z)
    @nospecialize x y
    ...
end

@nospecialize
f(y) = [x for x in y]
@specialize
```

`Base.@specialize` — Macro

```
@specialize
```

Reset the specialization hint for an argument back to the default. For details, see `@nospecialize`.

`Base.gensym` — Function

```
gensym([tag])
```

Generates a symbol which will not conflict with other variable names.

`Base.@gensym` — Macro

```
@gensym
```

Generates a gensym symbol for a variable. For example, `@gensym x y` is transformed into `x = gensym("x"); y = gensym("y")`.

`var"name"` — Keyword

```
var
```

The syntax `var"#example#"` refers to a variable named `Symbol("#example#")`, even though `#example#` is not a valid Julia identifier name.

This can be useful for interoperability with programming languages which have different rules for the construction of valid identifiers. For example, to refer to the `R` variable `draw.segments`, you can use `var"draw.segments"` in your Julia code.

It is also used to `show` julia source code which has gone through macro hygiene or otherwise contains variable names which can't be parsed normally.

Note that this syntax requires parser support so it is expanded directly by the parser rather than being implemented as a normal string macro `@var_str`.

> **Julia 1.3**
>
> This syntax requires at least Julia 1.3.

`Base.@goto` — Macro

```
@goto name
```

`@goto name` unconditionally jumps to the statement at the location `@label name`.

`@label` and `@goto` cannot create jumps to different top-level statements. Attempts cause an error. To still use `@goto`, enclose the `@label` and `@goto` in a block.

`Base.@label` — Macro

```
@label name
```

Labels a statement with the symbolic label `name`. The label marks the end-point of an unconditional jump with `@goto name`.

`Base.SimdLoop.@simd` — Macro

```
@simd
```

Annotate a `for` loop to allow the compiler to take extra liberties to allow loop re-ordering

> **Warning**
>
> This feature is experimental and could change or disappear in future versions of Julia. Incorrect use of the `@simd` macro may cause unexpected results.

The object iterated over in a `@simd for` loop should be a one-dimensional range. By using `@simd`, you are asserting several properties of the loop:

- It is safe to execute iterations in arbitrary or overlapping order, with special consideration for reduction variables.
- Floating-point operations on reduction variables can be reordered, possibly causing different results than without `@simd`.

In many cases, Julia is able to automatically vectorize inner for loops without the use of `@simd`. Using `@simd` gives the compiler a little extra leeway to make it possible in more situations. In

either case, your inner loop should have the following properties to allow vectorization:

- The loop must be an innermost loop
- The loop body must be straight-line code. Therefore, `@inbounds` is currently needed for all array accesses. The compiler can sometimes turn short `&&`, `||`, and `?:` expressions into straight-line code if it is safe to evaluate all operands unconditionally. Consider using the `ifelse` function instead of `?:` in the loop if it is safe to do so.
- Accesses must have a stride pattern and cannot be "gathers" (random-index reads) or "scatters" (random-index writes).
- The stride should be unit stride.

> **Note**
>
> The `@simd` does not assert by default that the loop is completely free of loop-carried memory dependencies, which is an assumption that can easily be violated in generic code. If you are writing non-generic code, you can use `@simd ivdep for ... end` to also assert that:

- There exists no loop-carried memory dependencies
- No iteration ever waits on a previous iteration to make forward progress.

`Base.@polly` — Macro

```
@polly
```

Tells the compiler to apply the polyhedral optimizer Polly to a function.

`Base.@generated` — Macro

```
@generated f
@generated(f)
```

`@generated` is used to annotate a function which will be generated. In the body of the generated function, only types of arguments can be read (not the values). The function returns a quoted expression evaluated when the function is called. The `@generated` macro should not be used on

functions mutating the global scope or depending on mutable elements.

See Metaprogramming for further details.

Example:

```
julia> @generated function bar(x)
           if x <: Integer
               return :(x ^ 2)
           else
               return :(x)
           end
       end
bar (generic function with 1 method)

julia> bar(4)
16

julia> bar("baz")
"baz"
```

Base.@pure — Macro

```
@pure ex
@pure(ex)
```

`@pure` gives the compiler a hint for the definition of a pure function, helping for type inference.

A pure function can only depend on immutable information. This also means a `@pure` function cannot use any global mutable state, including generic functions. Calls to generic functions depend on method tables which are mutable global state. Use with caution, incorrect `@pure` annotation of a function may introduce hard to identify bugs. Double check for calls to generic functions. This macro is intended for internal compiler use and may be subject to changes.

Base.@deprecate — Macro

```
@deprecate old new [ex=true]
```

Deprecate method `old` and specify the replacement call `new`. Prevent `@deprecate` from

exporting `old` by setting `ex` to `false`. `@deprecate` defines a new method with the same signature as `old`.

> **Julia 1.5**
>
> As of Julia 1.5, functions defined by `@deprecate` do not print warning when `julia` is run without the `--depwarn=yes` flag set, as the default value of `--depwarn` option is `no`. The warnings are printed from tests run by `Pkg.test()`.

Examples

```
julia> @deprecate old(x) new(x)
old (generic function with 1 method)

julia> @deprecate old(x) new(x) false
old (generic function with 1 method)
```

Missing Values

`Base.Missing` — Type

```
Missing
```

A type with no fields whose singleton instance `missing` is used to represent missing values.

`Base.missing` — Constant

```
missing
```

The singleton instance of type `Missing` representing a missing value.

`Base.coalesce` — Function

```
coalesce(x, y...)
```

Return the first value in the arguments which is not equal to `missing`, if any. Otherwise return `missing`.

See also `something`.

Examples

```
julia> coalesce(missing, 1)
1

julia> coalesce(1, missing)
1

julia> coalesce(nothing, 1)  # returns `nothing`

julia> coalesce(missing, missing)
missing
```

`Base.ismissing` — Function

```
ismissing(x)
```

Indicate whether `x` is `missing`.

`Base.skipmissing` — Function

```
skipmissing(itr)
```

Return an iterator over the elements in `itr` skipping `missing` values. The returned object can be indexed using indices of `itr` if the latter is indexable. Indices corresponding to missing values are not valid: they are skipped by `keys` and `eachindex`, and a `MissingException` is thrown when trying to use them.

Use `collect` to obtain an `Array` containing the non-`missing` values in `itr`. Note that even if `itr` is a multidimensional array, the result will always be a `Vector` since it is not possible to remove

missings while preserving dimensions of the input.

Examples

```
julia> x = skipmissing([1, missing, 2])
skipmissing(Union{Missing, Int64}[1, missing, 2])

julia> sum(x)
3

julia> x[1]
1

julia> x[2]
ERROR: MissingException: the value at index (2,) is missing
[...]

julia> argmax(x)
3

julia> collect(keys(x))
2-element Array{Int64,1}:
 1
 3

julia> collect(skipmissing([1, missing, 2]))
2-element Array{Int64,1}:
 1
 2

julia> collect(skipmissing([1 missing; 2 missing]))
2-element Array{Int64,1}:
 1
 2
```

`Base.nonmissingtype` — Function

```
nonmissingtype(T::Type)
```

If `T` is a union of types containing `Missing`, return a new type with `Missing` removed.

Examples

```
julia> nonmissingtype(Union{Int64,Missing})
Int64

julia> nonmissingtype(Any)
Any
```

> Julia 1.3

This function is exported as of Julia 1.3.

System

`Base.run` — Function

```
run(command, args...; wait::Bool = true)
```

Run a command object, constructed with backticks (see the Running External Programs section in the manual). Throws an error if anything goes wrong, including the process exiting with a non-zero status (when `wait` is true).

If `wait` is false, the process runs asynchronously. You can later wait for it and check its exit status by calling `success` on the returned process object.

When `wait` is false, the process' I/O streams are directed to `devnull`. When `wait` is true, I/O streams are shared with the parent process. Use `pipeline` to control I/O redirection.

`Base.devnull` — Constant

```
devnull
```

Used in a stream redirect to discard all data written to it. Essentially equivalent to `/dev/null` on Unix or `NUL` on Windows. Usage:

```
run(pipeline(`cat test.txt`, devnull))
```

`Base.success` — Function

```
success(command)
```

Run a command object, constructed with backticks (see the Running External Programs section in the manual), and tell whether it was successful (exited with a code of 0). An exception is raised if the process cannot be started.

`Base.process_running` — Function

```
process_running(p::Process)
```

Determine whether a process is currently running.

`Base.process_exited` — Function

```
process_exited(p::Process)
```

Determine whether a process has exited.

`Base.kill` — Method

```
kill(p::Process, signum=Base.SIGTERM)
```

Send a signal to a process. The default is to terminate the process. Returns successfully if the

process has already exited, but throws an error if killing the process failed for other reasons (e.g. insufficient permissions).

`Base.Sys.set_process_title` — Function

```
Sys.set_process_title(title::AbstractString)
```

Set the process title. No-op on some operating systems.

`Base.Sys.get_process_title` — Function

```
Sys.get_process_title()
```

Get the process title. On some systems, will always return an empty string.

`Base.ignorestatus` — Function

```
ignorestatus(command)
```

Mark a command object so that running it will not throw an error if the result code is non-zero.

`Base.detach` — Function

```
detach(command)
```

Mark a command object so that it will be run in a new process group, allowing it to outlive the julia process, and not have Ctrl-C interrupts passed to it.

`Base.Cmd` — Type

```
Cmd(cmd::Cmd; ignorestatus, detach, windows_verbatim, windows_hide, env, dir)
```

Construct a new `Cmd` object, representing an external program and arguments, from `cmd`, while changing the settings of the optional keyword arguments:

- `ignorestatus::Bool`: If `true` (defaults to `false`), then the `Cmd` will not throw an error if the return code is nonzero.
- `detach::Bool`: If `true` (defaults to `false`), then the `Cmd` will be run in a new process group, allowing it to outlive the `julia` process and not have Ctrl-C passed to it.
- `windows_verbatim::Bool`: If `true` (defaults to `false`), then on Windows the `Cmd` will send a command-line string to the process with no quoting or escaping of arguments, even arguments containing spaces. (On Windows, arguments are sent to a program as a single "command-line" string, and programs are responsible for parsing it into arguments. By default, empty arguments and arguments with spaces or tabs are quoted with double quotes `"` in the command line, and `\` or `"` are preceded by backslashes. `windows_verbatim=true` is useful for launching programs that parse their command line in nonstandard ways.) Has no effect on non-Windows systems.
- `windows_hide::Bool`: If `true` (defaults to `false`), then on Windows no new console window is displayed when the `Cmd` is executed. This has no effect if a console is already open or on non-Windows systems.
- `env`: Set environment variables to use when running the `Cmd`. `env` is either a dictionary mapping strings to strings, an array of strings of the form `"var=val"`, an array or tuple of `"var"=>val` pairs, or `nothing`. In order to modify (rather than replace) the existing environment, create `env` by `copy(ENV)` and then set `env["var"]=val` as desired.
- `dir::AbstractString`: Specify a working directory for the command (instead of the current directory).

For any keywords that are not specified, the current settings from `cmd` are used. Normally, to create a `Cmd` object in the first place, one uses backticks, e.g.

```
Cmd(`echo "Hello world"`, ignorestatus=true, detach=false)
```

`Base.setenv` — Function

```
setenv(command::Cmd, env; dir="")
```

Set environment variables to use when running the given `command`. `env` is either a dictionary mapping strings to strings, an array of strings of the form `"var=val"`, or zero or more `"var"=>val` pair arguments. In order to modify (rather than replace) the existing environment, create `env` by

`copy(ENV)` and then setting `env["var"]=val` as desired, or use `withenv`.

The `dir` keyword argument can be used to specify a working directory for the command.

`Base.withenv` — Function

```
withenv(f::Function, kv::Pair...)
```

Execute `f` in an environment that is temporarily modified (not replaced as in `setenv`) by zero or more `"var"=>val` arguments `kv`. `withenv` is generally used via the `withenv(kv...) do ... end` syntax. A value of `nothing` can be used to temporarily unset an environment variable (if it is set). When `withenv` returns, the original environment has been restored.

`Base.pipeline` — Method

```
pipeline(from, to, ...)
```

Create a pipeline from a data source to a destination. The source and destination can be commands, I/O streams, strings, or results of other `pipeline` calls. At least one argument must be a command. Strings refer to filenames. When called with more than two arguments, they are chained together from left to right. For example, `pipeline(a,b,c)` is equivalent to `pipeline(pipeline(a,b),c)`. This provides a more concise way to specify multi-stage pipelines.

Examples:

```
run(pipeline(`ls`, `grep xyz`))
run(pipeline(`ls`, "out.txt"))
run(pipeline("out.txt", `grep xyz`))
```

`Base.pipeline` — Method

```
pipeline(command; stdin, stdout, stderr, append=false)
```

Redirect I/O to or from the given `command`. Keyword arguments specify which of the command's streams should be redirected. `append` controls whether file output appends to the file. This is a

more general version of the 2-argument `pipeline` function. `pipeline(from, to)` is equivalent to `pipeline(from, stdout=to)` when `from` is a command, and to `pipeline(to, stdin=from)` when `from` is another kind of data source.

Examples:

```
run(pipeline(`dothings`, stdout="out.txt", stderr="errs.txt"))
run(pipeline(`update`, stdout="log.txt", append=true))
```

`Base.Libc.gethostname` — Function

```
gethostname() -> AbstractString
```

Get the local machine's host name.

`Base.Libc.getpid` — Function

```
getpid() -> Int32
```

Get Julia's process ID.

```
getpid(process) -> Int32
```

Get the child process ID, if it still exists.

> **Julia 1.1**
>
> This function requires at least Julia 1.1.

`Base.Libc.time` — Method

```
time()
```

Get the system time in seconds since the epoch, with fairly high (typically, microsecond) resolution.

`Base.time_ns` — Function

```
time_ns()
```

Get the time in nanoseconds. The time corresponding to 0 is undefined, and wraps every 5.8 years.

`Base.@time` — Macro

```
@time
```

A macro to execute an expression, printing the time it took to execute, the number of allocations, and the total number of bytes its execution caused to be allocated, before returning the value of the expression.

See also `@timev`, `@timed`, `@elapsed`, and `@allocated`.

> **Note**
>
> For more serious benchmarking, consider the `@btime` macro from the BenchmarkTools.jl package which among other things evaluates the function multiple times in order to reduce noise.

```
julia> @time rand(10^6);
  0.001525 seconds (7 allocations: 7.630 MiB)

julia> @time begin
           sleep(0.3)
           1+1
       end
  0.301395 seconds (8 allocations: 336 bytes)
2
```

`Base.@timev` — Macro

```
@timev
```

This is a verbose version of the `@time` macro. It first prints the same information as `@time`, then any non-zero memory allocation counters, and then returns the value of the expression.

See also `@time`, `@timed`, `@elapsed`, and `@allocated`.

```
julia> @timev rand(10^6);
  0.001006 seconds (7 allocations: 7.630 MiB)
elapsed time (ns): 1005567
bytes allocated:   8000256
pool allocs:       6
malloc() calls:    1
```

`Base.@timed` — Macro

```
@timed
```

A macro to execute an expression, and return the value of the expression, elapsed time, total bytes allocated, garbage collection time, and an object with various memory allocation counters.

See also `@time`, `@timev`, `@elapsed`, and `@allocated`.

```
julia> stats = @timed rand(10^6);

julia> stats.time
0.006634834

julia> stats.bytes
8000256

julia> stats.gctime
0.0055765

julia> propertynames(stats.gcstats)
(:allocd, :malloc, :realloc, :poolalloc, :bigalloc, :freecall, :total_time, :pa
```

```
julia> stats.gcstats.total_time
5576500
```

Julia 1.5

The return type of this macro was changed from `Tuple` to `NamedTuple` in Julia 1.5.

`Base.@elapsed` — Macro

```
@elapsed
```

A macro to evaluate an expression, discarding the resulting value, instead returning the number of seconds it took to execute as a floating-point number.

See also `@time`, `@timev`, `@timed`, and `@allocated`.

```
julia> @elapsed sleep(0.3)
0.301391426
```

`Base.@allocated` — Macro

```
@allocated
```

A macro to evaluate an expression, discarding the resulting value, instead returning the total number of bytes allocated during evaluation of the expression.

See also `@time`, `@timev`, `@timed`, and `@elapsed`.

```
julia> @allocated rand(10^6)
8000080
```

`Base.EnvDict` — Type

```
EnvDict() -> EnvDict
```

A singleton of this type provides a hash table interface to environment variables.

`Base.ENV` — Constant

```
ENV
```

Reference to the singleton `EnvDict`, providing a dictionary interface to system environment variables.

(On Windows, system environment variables are case-insensitive, and `ENV` correspondingly converts all keys to uppercase for display, iteration, and copying. Portable code should not rely on the ability to distinguish variables by case, and should beware that setting an ostensibly lowercase variable may result in an uppercase `ENV` key.)

`Base.Sys.isunix` — Function

```
Sys.isunix([os])
```

Predicate for testing if the OS provides a Unix-like interface. See documentation in Handling Operating System Variation.

`Base.Sys.isapple` — Function

```
Sys.isapple([os])
```

Predicate for testing if the OS is a derivative of Apple Macintosh OS X or Darwin. See documentation in Handling Operating System Variation.

`Base.Sys.islinux` — Function

```
Sys.islinux([os])
```

Predicate for testing if the OS is a derivative of Linux. See documentation in Handling Operating System Variation.

`Base.Sys.isbsd` — Function

```
Sys.isbsd([os])
```

Predicate for testing if the OS is a derivative of BSD. See documentation in Handling Operating System Variation.

> **Note**
>
> The Darwin kernel descends from BSD, which means that `Sys.isbsd()` is `true` on macOS systems. To exclude macOS from a predicate, use `Sys.isbsd() && !Sys.isapple()`.

`Base.Sys.isfreebsd` — Function

```
Sys.isfreebsd([os])
```

Predicate for testing if the OS is a derivative of FreeBSD. See documentation in Handling Operating System Variation.

> **Note**
>
> Not to be confused with `Sys.isbsd()`, which is `true` on FreeBSD but also on other BSD-based systems. `Sys.isfreebsd()` refers only to FreeBSD.

> **Julia 1.1**
>
> This function requires at least Julia 1.1.

`Base.Sys.isopenbsd` — Function

```
Sys.isopenbsd([os])
```

Predicate for testing if the OS is a derivative of OpenBSD. See documentation in Handling Operating System Variation.

> **Note**
>
> Not to be confused with `Sys.isbsd()`, which is `true` on OpenBSD but also on other BSD-based systems. `Sys.isopenbsd()` refers only to OpenBSD.

> **Julia 1.1**
>
> This function requires at least Julia 1.1.

`Base.Sys.isnetbsd` — Function

```
Sys.isnetbsd([os])
```

Predicate for testing if the OS is a derivative of NetBSD. See documentation in Handling Operating System Variation.

> **Note**
>
> Not to be confused with `Sys.isbsd()`, which is `true` on NetBSD but also on other BSD-based systems. `Sys.isnetbsd()` refers only to NetBSD.

> **Julia 1.1**
>
> This function requires at least Julia 1.1.

`Base.Sys.isdragonfly` — Function

```
Sys.isdragonfly([os])
```

Predicate for testing if the OS is a derivative of DragonFly BSD. See documentation in Handling Operating System Variation.

> **Note**
>
> Not to be confused with `Sys.isbsd()`, which is `true` on DragonFly but also on other BSD-based systems. `Sys.isdragonfly()` refers only to DragonFly.

> **Julia 1.1**
>
> This function requires at least Julia 1.1.

`Base.Sys.iswindows` — Function

```
Sys.iswindows([os])
```

Predicate for testing if the OS is a derivative of Microsoft Windows NT. See documentation in Handling Operating System Variation.

`Base.Sys.windows_version` — Function

```
Sys.windows_version()
```

Return the version number for the Windows NT Kernel as a `VersionNumber`, i.e. `v"major.minor.build"`, or `v"0.0.0"` if this is not running on Windows.

`Base.Sys.free_memory` — Function

```
Sys.free_memory()
```

Get the total free memory in RAM in bytes.

`Base.Sys.total_memory` — Function

```
Sys.total_memory()
```

Get the total memory in RAM (including that which is currently used) in bytes.

`Base.@static` — Macro

```
@static
```

Partially evaluate an expression at parse time.

For example, `@static Sys.iswindows() ? foo : bar` will evaluate `Sys.iswindows()` and insert either `foo` or `bar` into the expression. This is useful in cases where a construct would be invalid on other platforms, such as a `ccall` to a non-existent function. `@static if Sys.isapple() foo end` and `@static foo <&&, ||> bar` are also valid syntax.

Versioning

`Base.VersionNumber` — Type

```
VersionNumber
```

Version number type which follow the specifications of semantic versioning, composed of major, minor and patch numeric values, followed by pre-release and build alpha-numeric annotations. See also `@v_str`.

Examples

```
julia> VersionNumber("1.2.3")
```

```
v"1.2.3"

julia> VersionNumber("2.0.1-rc1")
v"2.0.1-rc1"
```

`Base.@v_str` — Macro

```
@v_str
```

String macro used to parse a string to a `VersionNumber`.

Examples

```
julia> v"1.2.3"
v"1.2.3"

julia> v"2.0.1-rc1"
v"2.0.1-rc1"
```

Errors

`Base.error` — Function

```
error(message::AbstractString)
```

Raise an `ErrorException` with the given message.

```
error(msg...)
```

Raise an `ErrorException` with the given message.

`Core.throw` — Function

```
throw(e)
```

Throw an object as an exception.

Base.rethrow — Function

```
rethrow()
```

Rethrow the current exception from within a `catch` block. The rethrown exception will continue propagation as if it had not been caught.

> **Note**
>
> The alternative form `rethrow(e)` allows you to associate an alternative exception object `e` with the current backtrace. However this misrepresents the program state at the time of the error so you're encouraged to instead throw a new exception using `throw(e)`. In Julia 1.1 and above, using `throw(e)` will preserve the root cause exception on the stack, as described in `catch_stack`.

Base.backtrace — Function

```
backtrace()
```

Get a backtrace object for the current program point.

Base.catch_backtrace — Function

```
catch_backtrace()
```

Get the backtrace of the current exception, for use within `catch` blocks.

`Base.catch_stack` — Function

```
catch_stack(task=current_task(); [inclue_bt=true])
```

Get the stack of exceptions currently being handled. For nested catch blocks there may be more than one current exception in which case the most recently thrown exception is last in the stack. The stack is returned as a Vector of `(exception,backtrace)` pairs, or a Vector of exceptions if `include_bt` is false.

Explicitly passing `task` will return the current exception stack on an arbitrary task. This is useful for inspecting tasks which have failed due to uncaught exceptions.

> **Julia 1.1**
>
> This function is experimental in Julia 1.1 and will likely be renamed in a future release (see https://github.com/JuliaLang/julia/pull/29901).

`Base.@assert` — Macro

```
@assert cond [text]
```

Throw an `AssertionError` if `cond` is `false`. Preferred syntax for writing assertions. Message `text` is optionally displayed upon assertion failure.

> **Warning**
>
> An assert might be disabled at various optimization levels. Assert should therefore only be used as a debugging tool and not used for authentication verification (e.g., verifying passwords), nor should side effects needed for the function to work correctly be used inside of asserts.

Examples

```
julia> @assert iseven(3) "3 is an odd number!"
ERROR: AssertionError: 3 is an odd number!
```

```
julia> @assert isodd(3) "What even are numbers?"
```

Base.Experimental.register_error_hint — Function

```
Experimental.register_error_hint(handler, exceptiontype)
```

Register a "hinting" function `handler(io, exception)` that can suggest potential ways for users to circumvent errors. `handler` should examine `exception` to see whether the conditions appropriate for a hint are met, and if so generate output to `io`. Packages should call `register_error_hint` from within their `__init__` function.

For specific exception types, `handler` is required to accept additional arguments:

- `MethodError`: provide `handler(io, exc::MethodError, argtypes, kwargs)`, which splits the combined arguments into positional and keyword arguments.

When issuing a hint, the output should typically start with `\n`.

If you define custom exception types, your `showerror` method can support hints by calling `Experimental.show_error_hints`.

Example

```
julia> module Hinter

       only_int(x::Int)      = 1
       any_number(x::Number) = 2

       function __init__()
           Base.Experimental.register_error_hint(MethodError) do io, exc, argtyp
               if exc.f == only_int
                    # Color is not necessary, this is just to show it's possible
                    print(io, "\nDid you mean to call ")
                    printstyled(io, "`any_number`?", color=:cyan)
               end
           end
       end

       end
```

Then if you call `Hinter.only_int` on something that isn't an `Int` (thereby triggering a

`MethodError`), it issues the hint:

```
julia> Hinter.only_int(1.0)
ERROR: MethodError: no method matching only_int(::Float64)
Did you mean to call `any_number`?
Closest candidates are:
    ...
```

> **Julia 1.5**
>
> Custom error hints are available as of Julia 1.5.

> **Warning**
>
> This interface is experimental and subject to change or removal without notice. To insulate yourself against changes, consider putting any registrations inside an `if isdefined(Base.Experimental, :register_error_hint) ... end` block.

`Base.Experimental.show_error_hints` — Function

```
Experimental.show_error_hints(io, ex, args...)
```

Invoke all handlers from `Experimental.register_error_hint` for the particular exception type `typeof(ex)`. `args` must contain any other arguments expected by the handler for that type.

> **Julia 1.5**
>
> Custom error hints are available as of Julia 1.5.

> **Warning**
>
> This interface is experimental and subject to change or removal without notice.

Core.ArgumentError — Type

```
ArgumentError(msg)
```

The parameters to a function call do not match a valid signature. Argument `msg` is a descriptive error string.

Core.AssertionError — Type

```
AssertionError([msg])
```

The asserted condition did not evaluate to `true`. Optional argument `msg` is a descriptive error string.

Examples

```
julia> @assert false "this is not true"
ERROR: AssertionError: this is not true
```

`AssertionError` is usually thrown from `@assert`.

Core.BoundsError — Type

```
BoundsError([a],[i])
```

An indexing operation into an array, `a`, tried to access an out-of-bounds element at index `i`.

Examples

```
julia> A = fill(1.0, 7);

julia> A[8]
ERROR: BoundsError: attempt to access 7-element Array{Float64,1} at index [8]
Stacktrace:
 [1] getindex(::Array{Float64,1}, ::Int64) at ./array.jl:660
 [2] top-level scope
```

```
julia> B = fill(1.0, (2,3));

julia> B[2, 4]
ERROR: BoundsError: attempt to access 2×3 Array{Float64,2} at index [2, 4]
Stacktrace:
 [1] getindex(::Array{Float64,2}, ::Int64, ::Int64) at ./array.jl:661
 [2] top-level scope

julia> B[9]
ERROR: BoundsError: attempt to access 2×3 Array{Float64,2} at index [9]
Stacktrace:
 [1] getindex(::Array{Float64,2}, ::Int64) at ./array.jl:660
 [2] top-level scope
```

Base.CompositeException — Type

```
CompositeException
```

Wrap a `Vector` of exceptions thrown by a `Task` (e.g. generated from a remote worker over a channel or an asynchronously executing local I/O write or a remote worker under `pmap`) with information about the series of exceptions. For example, if a group of workers are executing several tasks, and multiple workers fail, the resulting `CompositeException` will contain a "bundle" of information from each worker indicating where and why the exception(s) occurred.

Base.DimensionMismatch — Type

```
DimensionMismatch([msg])
```

The objects called do not have matching dimensionality. Optional argument `msg` is a descriptive error string.

Core.DivideError — Type

```
DivideError()
```

Integer division was attempted with a denominator value of 0.

Examples

```
julia> 2/0
Inf

julia> div(2, 0)
ERROR: DivideError: integer division error
Stacktrace:
[...]
```

Core.DomainError — Type

```
DomainError(val)
DomainError(val, msg)
```

The argument `val` to a function or constructor is outside the valid domain.

Examples

```
julia> sqrt(-1)
ERROR: DomainError with -1.0:
sqrt will only return a complex result if called with a complex argument. Try s
Stacktrace:
[...]
```

Base.EOFError — Type

```
EOFError()
```

No more data was available to read from a file or stream.

Core.ErrorException — Type

```
ErrorException(msg)
```

Generic error type. The error message, in the `.msg` field, may provide more specific details.

Examples

```
julia> ex = ErrorException("I've done a bad thing");

julia> ex.msg
"I've done a bad thing"
```

Core.InexactError — Type

```
InexactError(name::Symbol, T, val)
```

Cannot exactly convert `val` to type `T` in a method of function `name`.

Examples

```
julia> convert(Float64, 1+2im)
ERROR: InexactError: Float64(1 + 2im)
Stacktrace:
[...]
```

Core.InterruptException — Type

```
InterruptException()
```

The process was stopped by a terminal interrupt (CTRL+C).

Note that, in Julia script started without `-i` (interactive) option, `InterruptException` is not thrown by default. Calling `Base.exit_on_sigint(false)` in the script can recover the behavior of the REPL. Alternatively, a Julia script can be started with

```
julia -e "include(popfirst!(ARGS))" script.jl
```

to let `InterruptException` be thrown by CTRL+C during the execution.

Base.KeyError — Type

```
KeyError(key)
```

An indexing operation into an `AbstractDict` (`Dict`) or `Set` like object tried to access or delete a non-existent element.

`Core.LoadError` — Type

```
LoadError(file::AbstractString, line::Int, error)
```

An error occurred while `include`ing, `require`ing, or `using` a file. The error specifics should be available in the `.error` field.

`Core.MethodError` — Type

```
MethodError(f, args)
```

A method with the required type signature does not exist in the given generic function. Alternatively, there is no unique most-specific method.

`Base.MissingException` — Type

```
MissingException(msg)
```

Exception thrown when a `missing` value is encountered in a situation where it is not supported. The error message, in the `msg` field may provide more specific details.

`Core.OutOfMemoryError` — Type

```
OutOfMemoryError()
```

An operation allocated too much memory for either the system or the garbage collector to handle properly.

Core.ReadOnlyMemoryError — Type

```
ReadOnlyMemoryError()
```

An operation tried to write to memory that is read-only.

Core.OverflowError — Type

```
OverflowError(msg)
```

The result of an expression is too large for the specified type and will cause a wraparound.

Base.ProcessFailedException — Type

```
ProcessFailedException
```

Indicates problematic exit status of a process. When running commands or pipelines, this is thrown to indicate a nonzero exit code was returned (i.e. that the invoked process failed).

Core.StackOverflowError — Type

```
StackOverflowError()
```

The function call grew beyond the size of the call stack. This usually happens when a call recurses infinitely.

Base.SystemError — Type

```
SystemError(prefix::AbstractString, [errno::Int32])
```

A system call failed with an error code (in the `errno` global variable).

Core.TypeError — Type

```
TypeError(func::Symbol, context::AbstractString, expected::Type, got)
```

A type assertion failure, or calling an intrinsic function with an incorrect argument type.

Core.UndefKeywordError — Type

```
UndefKeywordError(var::Symbol)
```

The required keyword argument `var` was not assigned in a function call.

Examples

```
julia> function my_func(;my_arg)
           return my_arg + 1
       end
my_func (generic function with 1 method)

julia> my_func()
ERROR: UndefKeywordError: keyword argument my_arg not assigned
Stacktrace:
 [1] my_func() at ./REPL[1]:2
 [2] top-level scope at REPL[2]:1
```

Core.UndefRefError — Type

```
UndefRefError()
```

The item or field is not defined for the given object.

Examples

```
julia> struct MyType
           a::Vector{Int}
           MyType() = new()
       end
```

```
julia> A = MyType()
MyType(#undef)

julia> A.a
ERROR: UndefRefError: access to undefined reference
Stacktrace:
[...]
```

Core.UndefVarError — Type

```
UndefVarError(var::Symbol)
```

A symbol in the current scope is not defined.

Examples

```
julia> a
ERROR: UndefVarError: a not defined

julia> a = 1;

julia> a
1
```

Base.StringIndexError — Type

```
StringIndexError(str, i)
```

An error occurred when trying to access `str` at index `i` that is not valid.

Core.InitError — Type

```
InitError(mod::Symbol, error)
```

An error occurred when running a module's `__init__` function. The actual error thrown is

available in the `.error` field.

`Base.retry` — Function

```
retry(f;  delays=ExponentialBackOff(), check=nothing) -> Function
```

Return an anonymous function that calls function `f`. If an exception arises, `f` is repeatedly called again, each time `check` returns `true`, after waiting the number of seconds specified in `delays`. `check` should input `delays`'s current state and the `Exception`.

> **Julia 1.2**
>
> Before Julia 1.2 this signature was restricted to `f::Function`.

Examples

```
retry(f, delays=fill(5.0, 3))
retry(f, delays=rand(5:10, 2))
retry(f, delays=Base.ExponentialBackOff(n=3, first_delay=5, max_delay=1000))
retry(http_get, check=(s,e)->e.status == "503")(url)
retry(read, check=(s,e)->isa(e, IOError))(io, 128; all=false)
```

`Base.ExponentialBackOff` — Type

```
ExponentialBackOff(; n=1, first_delay=0.05, max_delay=10.0, factor=5.0, jitter=
```

A `Float64` iterator of length `n` whose elements exponentially increase at a rate in the interval `factor` * (1 ± `jitter`). The first element is `first_delay` and all elements are clamped to `max_delay`.

Events

`Base.Timer` — Method

```
Timer(callback::Function, delay; interval = 0)
```

Create a timer that wakes up tasks waiting for it (by calling `wait` on the timer object) and calls the function `callback`.

Waiting tasks are woken and the function `callback` is called after an initial delay of `delay` seconds, and then repeating with the given `interval` in seconds. If `interval` is equal to `0`, the timer is only triggered once. The function `callback` is called with a single argument, the timer itself. When the timer is closed (by `close` waiting tasks are woken with an error. Use `isopen` to check whether a timer is still active.

Examples

Here the first number is printed after a delay of two seconds, then the following numbers are printed quickly.

```
julia> begin
           i = 0
           cb(timer) = (global i += 1; println(i))
           t = Timer(cb, 2, interval=0.2)
           wait(t)
           sleep(0.5)
           close(t)
       end
1
2
3
```

`Base.Timer` — Type

```
Timer(delay; interval = 0)
```

Create a timer that wakes up tasks waiting for it (by calling `wait` on the timer object).

Waiting tasks are woken after an initial delay of `delay` seconds, and then repeating with the given `interval` in seconds. If `interval` is equal to `0`, the timer is only triggered once. When the timer is closed (by `close` waiting tasks are woken with an error. Use `isopen` to check whether a timer is still active.

`Base.AsyncCondition` — Type

```
AsyncCondition()
```

Create a async condition that wakes up tasks waiting for it (by calling `wait` on the object) when notified from C by a call to `uv_async_send`. Waiting tasks are woken with an error when the object is closed (by `close`. Use `isopen` to check whether it is still active.

`Base.AsyncCondition` — Method

```
AsyncCondition(callback::Function)
```

Create a async condition that calls the given `callback` function. The `callback` is passed one argument, the async condition object itself.

Reflection

`Base.nameof` — Method

```
nameof(m::Module) -> Symbol
```

Get the name of a `Module` as a `Symbol`.

Examples

```
julia> nameof(Base.Broadcast)
:Broadcast
```

`Base.parentmodule` — Function

```
parentmodule(m::Module) -> Module
```

Get a module's enclosing `Module`. `Main` is its own parent.

Examples

```
julia> parentmodule(Main)
Main

julia> parentmodule(Base.Broadcast)
Base
```

```
parentmodule(t::DataType) -> Module
```

Determine the module containing the definition of a (potentially `UnionAll`-wrapped) `DataType`.

Examples

```
julia> module Foo
           struct Int end
       end
Foo

julia> parentmodule(Int)
Core

julia> parentmodule(Foo.Int)
Foo
```

```
parentmodule(f::Function) -> Module
```

Determine the module containing the (first) definition of a generic function.

```
parentmodule(f::Function, types) -> Module
```

Determine the module containing a given definition of a generic function.

`Base.pathof` — Method

```
pathof(m::Module)
```

Return the path of the `m.jl` file that was used to `import` module `m`, or `nothing` if `m` was not imported from a package.

Use `dirname` to get the directory part and `basename` to get the file name part of the path.

`Base.moduleroot` — Function

```
moduleroot(m::Module) -> Module
```

Find the root module of a given module. This is the first module in the chain of parent modules of `m` which is either a registered root module or which is its own parent module.

`Base.@__MODULE__` — Macro

```
@__MODULE__ -> Module
```

Get the `Module` of the toplevel eval, which is the `Module` code is currently being read from.

`Base.fullname` — Function

```
fullname(m::Module)
```

Get the fully-qualified name of a module as a tuple of symbols. For example,

Examples

```
julia> fullname(Base.Iterators)
(:Base, :Iterators)

julia> fullname(Main)
(:Main,)
```

`Base.names` — Function

```
names(x::Module; all::Bool = false, imported::Bool = false)
```

Get an array of the names exported by a `Module`, excluding deprecated names. If `all` is true, then the list also includes non-exported names defined in the module, deprecated names, and compiler-generated names. If `imported` is true, then names explicitly imported from other modules are also included.

As a special case, all names defined in `Main` are considered "exported", since it is not idiomatic to explicitly export names from `Main`.

`Core.nfields` — Function

```
nfields(x) -> Int
```

Get the number of fields in the given object.

Examples

```
julia> a = 1//2;

julia> nfields(a)
2

julia> b = 1
1

julia> nfields(b)
0

julia> ex = ErrorException("I've done a bad thing");

julia> nfields(ex)
1
```

In these examples, `a` is a `Rational`, which has two fields. `b` is an `Int`, which is a primitive bitstype with no fields at all. `ex` is an `ErrorException`, which has one field.

`Base.isconst` — Function

```
isconst(m::Module, s::Symbol) -> Bool
```

Determine whether a global is declared `const` in a given `Module`.

`Base.nameof` — Method

```
nameof(f::Function) -> Symbol
```

Get the name of a generic `Function` as a symbol. For anonymous functions, this is a compiler-generated name. For explicitly-declared subtypes of `Function`, it is the name of the function's type.

`Base.functionloc` — Method

```
functionloc(f::Function, types)
```

Returns a tuple `(filename,line)` giving the location of a generic `Function` definition.

`Base.functionloc` — Method

```
functionloc(m::Method)
```

Returns a tuple `(filename,line)` giving the location of a `Method` definition.

Internals

`Base.GC.gc` — Function

```
GC.gc([full=true])
```

Perform garbage collection. The argument `full` determines the kind of collection: A full collection (default) sweeps all objects, which makes the next GC scan much slower, while an

incremental collection may only sweep so-called young objects.

> **Warning**
>
> Excessive use will likely lead to poor performance.

`Base.GC.enable` — Function

```
GC.enable(on::Bool)
```

Control whether garbage collection is enabled using a boolean argument (`true` for enabled, `false` for disabled). Return previous GC state.

> **Warning**
>
> Disabling garbage collection should be used only with caution, as it can cause memory use to grow without bound.

`Base.GC.@preserve` — Macro

```
GC.@preserve x1 x2 ... xn expr
```

Mark the objects `x1, x2, ...` as being *in use* during the evaluation of the expression `expr`. This is only required in unsafe code where `expr` *implicitly uses* memory or other resources owned by one of the `x`s.

Implicit use of `x` covers any indirect use of resources logically owned by `x` which the compiler cannot see. Some examples:

- Accessing memory of an object directly via a `Ptr`
- Passing a pointer to `x` to `ccall`
- Using resources of `x` which would be cleaned up in the finalizer.

`@preserve` should generally not have any performance impact in typical use cases where it briefly extends object lifetime. In implementation, `@preserve` has effects such as protecting dynamically

allocated objects from garbage collection.

Examples

When loading from a pointer with `unsafe_load`, the underlying object is implicitly used, for example `x` is implicitly used by `unsafe_load(p)` in the following:

```
julia> let
           x = Ref{Int}(101)
           p = Base.unsafe_convert(Ptr{Int}, x)
           GC.@preserve x unsafe_load(p)
       end
101
```

When passing pointers to `ccall`, the pointed-to object is implicitly used and should be preserved. (Note however that you should normally just pass `x` directly to `ccall` which counts as an explicit use.)

```
julia> let
           x = "Hello"
           p = pointer(x)
           Int(GC.@preserve x @ccall strlen(p::Cstring)::Csize_t)
           # Preferred alternative
           Int(@ccall strlen(x::Cstring)::Csize_t)
       end
5
```

`Base.GC.safepoint` — Function

```
GC.safepoint()
```

Inserts a point in the program where garbage collection may run. This can be useful in rare cases in multi-threaded programs where some threads are allocating memory (and hence may need to run GC) but other threads are doing only simple operations (no allocation, task switches, or I/O). Calling this function periodically in non-allocating threads allows garbage collection to run.

> **Julia 1.4**
>
> This function is available as of Julia 1.4.

`Base.Meta.lower` — Function

```
lower(m, x)
```

Takes the expression `x` and returns an equivalent expression in lowered form for executing in module `m`. See also `code_lowered`.

`Base.Meta.@lower` — Macro

```
@lower [m] x
```

Return lowered form of the expression `x` in module `m`. By default `m` is the module in which the macro is called. See also `lower`.

`Base.Meta.parse` — Method

```
parse(str, start; greedy=true, raise=true, depwarn=true)
```

Parse the expression string and return an expression (which could later be passed to eval for execution). `start` is the index of the first character to start parsing. If `greedy` is `true` (default), `parse` will try to consume as much input as it can; otherwise, it will stop as soon as it has parsed a valid expression. Incomplete but otherwise syntactically valid expressions will return

`Expr(:incomplete, "(error message)")`. If `raise` is `true` (default), syntax errors other than incomplete expressions will raise an error. If `raise` is `false`, `parse` will return an expression that will raise an error upon evaluation. If `depwarn` is `false`, deprecation warnings will be suppressed.

```
julia> Meta.parse("x = 3, y = 5", 7)
(:(y = 5), 13)

julia> Meta.parse("x = 3, y = 5", 5)
(:((3, y) = 5), 13)
```

`Base.Meta.parse` — Method

```
parse(str; raise=true, depwarn=true)
```

Parse the expression string greedily, returning a single expression. An error is thrown if there are additional characters after the first expression. If `raise` is `true` (default), syntax errors will raise an error; otherwise, `parse` will return an expression that will raise an error upon evaluation. If `depwarn` is `false`, deprecation warnings will be suppressed.

```
julia> Meta.parse("x = 3")
:(x = 3)

julia> Meta.parse("x = ")
:($(Expr(:incomplete, "incomplete: premature end of input")))

julia> Meta.parse("1.0.2")
ERROR: Base.Meta.ParseError("invalid numeric constant \"1.0.\"")
Stacktrace:
[...]

julia> Meta.parse("1.0.2"; raise = false)
:($(Expr(:error, "invalid numeric constant \"1.0.\"")))
```

`Base.Meta.ParseError` — Type

```
ParseError(msg)
```

The expression passed to the `parse` function could not be interpreted as a valid Julia expression.

`Core.QuoteNode` — Type

```
QuoteNode
```

A quoted piece of code, that does not support interpolation. See the manual section about QuoteNodes for details.

`Base.macroexpand` — Function

```
macroexpand(m::Module, x; recursive=true)
```

Take the expression `x` and return an equivalent expression with all macros removed (expanded) for executing in module `m`. The `recursive` keyword controls whether deeper levels of nested macros are also expanded. This is demonstrated in the example below:

```
julia> module M
           macro m1()
               42
           end
           macro m2()
               :(@m1())
           end
       end
M

julia> macroexpand(M, :(@m2()), recursive=true)
42

julia> macroexpand(M, :(@m2()), recursive=false)
:(#= REPL[16]:6 =# M.@m1)
```

`Base.@macroexpand` — Macro

```
@macroexpand
```

Return equivalent expression with all macros removed (expanded).

There are differences between `@macroexpand` and `macroexpand`.

- While `macroexpand` takes a keyword argument `recursive`, `@macroexpand` is always recursive. For a non recursive macro version, see `@macroexpand1`.
- While `macroexpand` has an explicit `module` argument, `@macroexpand` always expands with respect to the module in which it is called.

This is best seen in the following example:

```
julia> module M
           macro m()
               1
           end
           function f()
               (@macroexpand(@m),
                macroexpand(M, :(@m)),
                macroexpand(Main, :(@m))
               )
           end
       end
M

julia> macro m()
           2
       end
@m (macro with 1 method)

julia> M.f()
(1, 1, 2)
```

With `@macroexpand` the expression expands where `@macroexpand` appears in the code (module `M` in the example). With `macroexpand` the expression expands in the module given as the first argument.

`Base.@macroexpand1` — Macro

```
@macroexpand1
```

Non recursive version of `@macroexpand`.

`Base.code_lowered` — Function

```
code_lowered(f, types; generated=true, debuginfo=:default)
```

Return an array of the lowered forms (IR) for the methods matching the given generic function and type signature.

If `generated` is `false`, the returned `CodeInfo` instances will correspond to fallback implementations. An error is thrown if no fallback implementation exists. If `generated` is `true`, these `CodeInfo` instances will correspond to the method bodies yielded by expanding the generators.

The keyword `debuginfo` controls the amount of code metadata present in the output.

Note that an error will be thrown if `types` are not leaf types when `generated` is `true` and any of the corresponding methods are an `@generated` method.

`Base.code_typed` — Function

```
code_typed(f, types; optimize=true, debuginfo=:default)
```

Returns an array of type-inferred lowered form (IR) for the methods matching the given generic function and type signature. The keyword argument `optimize` controls whether additional optimizations, such as inlining, are also applied. The keyword `debuginfo` controls the amount of code metadata present in the output, possible options are `:source` or `:none`.

`Base.precompile` — Function

```
precompile(f, args::Tuple{Vararg{Any}})
```

Compile the given function `f` for the argument tuple (of types) `args`, but do not execute it.

Meta

`Base.Meta.quot` — Function

```
Meta.quot(ex)::Expr
```

Quote expression `ex` to produce an expression with head `quote`. This can for instance be used to represent objects of type `Expr` in the AST. See also the manual section about QuoteNode.

Examples

```
julia> eval(Meta.quot(:x))
:x

julia> dump(Meta.quot(:x))
Expr
  head: Symbol quote
  args: Array{Any}((1,))
    1: Symbol x

julia> eval(Meta.quot(:(1+2)))
:(1 + 2)
```

`Base.Meta.isexpr` — Function

```
Meta.isexpr(ex, head[, n])::Bool
```

Check if `ex` is an expression with head `head` and `n` arguments.

Examples

```
julia> ex = :(f(x))
```

```
:(f(x))

julia> Meta.isexpr(ex, :block)
false

julia> Meta.isexpr(ex, :call)
true

julia> Meta.isexpr(ex, [:block, :call]) # multiple possible heads
true

julia> Meta.isexpr(ex, :call, 1)
false

julia> Meta.isexpr(ex, :call, 2)
true
```

`Base.Meta.show_sexpr` — Function

```
Meta.show_sexpr([io::IO,], ex)
```

Show expression `ex` as a lisp style S-expression.

Examples

```
julia> Meta.show_sexpr(:(f(x, g(y,z))))
(:call, :f, :x, (:call, :g, :y, :z))
```

« Unicode Input Collections and Data Structures »