Base / Arrays Edit on GitHub

Arrays

Constructors and Types

`Core.AbstractArray` — Type

```
AbstractArray{T,N}
```

Supertype for `N`-dimensional arrays (or array-like types) with elements of type `T`. `Array` and other types are subtypes of this. See the manual section on the `AbstractArray` interface.

`Base.AbstractVector` — Type

```
AbstractVector{T}
```

Supertype for one-dimensional arrays (or array-like types) with elements of type `T`. Alias for `AbstractArray{T,1}`.

`Base.AbstractMatrix` — Type

```
AbstractMatrix{T}
```

Supertype for two-dimensional arrays (or array-like types) with elements of type `T`. Alias for `AbstractArray{T,2}`.

`Base.AbstractVecOrMat` — Constant

```
AbstractVecOrMat{T}
```

Union type of `AbstractVector{T}` and `AbstractMatrix{T}`.

`Core.Array` — Type

```
Array{T,N} <: AbstractArray{T,N}
```

`N`-dimensional dense array with elements of type `T`.

`Core.Array` — Method

```
Array{T}(undef, dims)
Array{T,N}(undef, dims)
```

Construct an uninitialized `N`-dimensional `Array` containing elements of type `T`. `N` can either be supplied explicitly, as in `Array{T,N}(undef, dims)`, or be determined by the length or number of `dims`. `dims` may be a tuple or a series of integer arguments corresponding to the lengths in each dimension. If the rank `N` is supplied explicitly, then it must match the length or number of `dims`. See `undef`.

Examples

```
julia> A = Array{Float64,2}(undef, 2, 3) # N given explicitly
2×3 Array{Float64,2}:
 6.90198e-310  6.90198e-310  6.90198e-310
 6.90198e-310  6.90198e-310  0.0

julia> B = Array{Float64}(undef, 2) # N determined by the input
2-element Array{Float64,1}:
 1.87103e-320
 0.0
```

`Core.Array` — Method

```
Array{T}(nothing, dims)
Array{T,N}(nothing, dims)
```

Construct an `N`-dimensional `Array` containing elements of type `T`, initialized with `nothing` entries. Element type `T` must be able to hold these values, i.e. `Nothing <: T`.

Examples

```
julia> Array{Union{Nothing, String}}(nothing, 2)
2-element Array{Union{Nothing, String},1}:
 nothing
 nothing

julia> Array{Union{Nothing, Int}}(nothing, 2, 3)
2×3 Array{Union{Nothing, Int64},2}:
 nothing  nothing  nothing
 nothing  nothing  nothing
```

`Core.Array` — Method

```
Array{T}(missing, dims)
Array{T,N}(missing, dims)
```

Construct an `N`-dimensional `Array` containing elements of type `T`, initialized with `missing` entries. Element type `T` must be able to hold these values, i.e. `Missing <: T`.

Examples

```
julia> Array{Union{Missing, String}}(missing, 2)
2-element Array{Union{Missing, String},1}:
 missing
 missing

julia> Array{Union{Missing, Int}}(missing, 2, 3)
2×3 Array{Union{Missing, Int64},2}:
 missing  missing  missing
 missing  missing  missing
```

`Core.UndefInitializer` — Type

```
UndefInitializer
```

Singleton type used in array initialization, indicating the array-constructor-caller would like an uninitialized array. See also `undef`, an alias for `UndefInitializer()`.

Examples

```
julia> Array{Float64,1}(UndefInitializer(), 3)
3-element Array{Float64,1}:
 2.2752528595e-314
 2.202942107e-314
 2.275252907e-314
```

`Core.undef` — Constant

```
undef
```

Alias for `UndefInitializer()`, which constructs an instance of the singleton type `UndefInitializer`, used in array initialization to indicate the array-constructor-caller would like an uninitialized array.

Examples

```
julia> Array{Float64,1}(undef, 3)
3-element Array{Float64,1}:
 2.2752528595e-314
 2.202942107e-314
 2.275252907e-314
```

`Base.Vector` — Type

```
Vector{T} <: AbstractVector{T}
```

One-dimensional dense array with elements of type `T`, often used to represent a mathematical vector. Alias for `Array{T,1}`.

`Base.Vector` — Method

```
Vector{T}(undef, n)
```

Construct an uninitialized `Vector{T}` of length `n`. See `undef`.

Examples

```
julia> Vector{Float64}(undef, 3)
3-element Array{Float64,1}:
 6.90966e-310
 6.90966e-310
 6.90966e-310
```

`Base.Vector` — Method

```
Vector{T}(nothing, m)
```

Construct a `Vector{T}` of length `m`, initialized with `nothing` entries. Element type `T` must be able to hold these values, i.e. `Nothing <: T`.

Examples

```
julia> Vector{Union{Nothing, String}}(nothing, 2)
2-element Array{Union{Nothing, String},1}:
 nothing
 nothing
```

`Base.Vector` — Method

```
Vector{T}(missing, m)
```

Construct a `Vector{T}` of length `m`, initialized with `missing` entries. Element type `T` must be able to hold these values, i.e. `Missing <: T`.

Examples

```
julia> Vector{Union{Missing, String}}(missing, 2)
2-element Array{Union{Missing, String},1}:
 missing
 missing
```

`Base.Matrix` — Type

```
Matrix{T} <: AbstractMatrix{T}
```

Two-dimensional dense array with elements of type `T`, often used to represent a mathematical matrix. Alias for `Array{T,2}`.

`Base.Matrix` — Method

```
Matrix{T}(undef, m, n)
```

Construct an uninitialized `Matrix{T}` of size `m×n`. See `undef`.

Examples

```
julia> Matrix{Float64}(undef, 2, 3)
2×3 Array{Float64,2}:
 6.93517e-310  6.93517e-310  6.93517e-310
 6.93517e-310  6.93517e-310  1.29396e-320
```

`Base.Matrix` — Method

```
Matrix{T}(nothing, m, n)
```

Construct a `Matrix{T}` of size `m×n`, initialized with `nothing` entries. Element type `T` must be able to hold these values, i.e. `Nothing <: T`.

Examples

```
julia> Matrix{Union{Nothing, String}}(nothing, 2, 3)
2×3 Array{Union{Nothing, String},2}:
 nothing  nothing  nothing
 nothing  nothing  nothing
```

`Base.Matrix` — Method

```
Matrix{T}(missing, m, n)
```

Construct a `Matrix{T}` of size `m×n`, initialized with `missing` entries. Element type `T` must be able to hold these values, i.e. `Missing <: T`.

Examples

```
julia> Matrix{Union{Missing, String}}(missing, 2, 3)
2×3 Array{Union{Missing, String},2}:
 missing  missing  missing
 missing  missing  missing
```

`Base.VecOrMat` — Constant

```
VecOrMat{T}
```

Union type of `Vector{T}` and `Matrix{T}`.

`Core.DenseArray` — Type

```
DenseArray{T, N} <: AbstractArray{T,N}
```

`N`-dimensional dense array with elements of type `T`. The elements of a dense array are stored contiguously in memory.

`Base.DenseVector` — Type

```
DenseVector{T}
```

One-dimensional `DenseArray` with elements of type `T`. Alias for `DenseArray{T,1}`.

`Base.DenseMatrix` — Type

```
DenseMatrix{T}
```

Two-dimensional `DenseArray` with elements of type `T`. Alias for `DenseArray{T,2}`.

`Base.DenseVecOrMat` — Constant

```
DenseVecOrMat{T}
```

Union type of `DenseVector{T}` and `DenseMatrix{T}`.

`Base.StridedArray` — Constant

```
StridedArray{T, N}
```

A hard-coded `Union` of common array types that follow the strided array interface, with elements of type `T` and `N` dimensions.

If `A` is a `StridedArray`, then its elements are stored in memory with offsets, which may vary between dimensions but are constant within a dimension. For example, `A` could have stride 2 in dimension 1, and stride 3 in dimension 2. Incrementing `A` along dimension `d` jumps in memory by [`strides(A, d)`] slots. Strided arrays are particularly important and useful because they can sometimes be passed directly as pointers to foreign language libraries like BLAS.

`Base.StridedVector` — Constant

```
StridedVector{T}
```

One dimensional `StridedArray` with elements of type `T`.

`Base.StridedMatrix` — Constant

```
StridedMatrix{T}
```

Two dimensional `StridedArray` with elements of type `T`.

`Base.StridedVecOrMat` — Constant

```
StridedVecOrMat{T}
```

Union type of `StridedVector` and `StridedMatrix` with elements of type `T`.

`Base.getindex` — Method

```
getindex(type[, elements...])
```

Construct a 1-d array of the specified type. This is usually called with the syntax `Type[]`. Element values can be specified using `Type[a,b,c,...]`.

Examples

```
julia> Int8[1, 2, 3]
3-element Array{Int8,1}:
 1
 2
 3

julia> getindex(Int8, 1, 2, 3)
3-element Array{Int8,1}:
 1
 2
```

```
3
```

`Base.zeros` — Function

```
zeros([T=Float64,] dims::Tuple)
zeros([T=Float64,] dims...)
```

Create an `Array`, with element type `T`, of all zeros with size specified by `dims`. See also `fill`, `ones`.

Examples

```
julia> zeros(1)
1-element Array{Float64,1}:
 0.0

julia> zeros(Int8, 2, 3)
2×3 Array{Int8,2}:
 0  0  0
 0  0  0
```

`Base.ones` — Function

```
ones([T=Float64,] dims::Tuple)
ones([T=Float64,] dims...)
```

Create an `Array`, with element type `T`, of all ones with size specified by `dims`. See also: `fill`, `zeros`.

Examples

```
julia> ones(1,2)
1×2 Array{Float64,2}:
 1.0  1.0

julia> ones(ComplexF64, 2, 3)
2×3 Array{Complex{Float64},2}:
 1.0+0.0im  1.0+0.0im  1.0+0.0im
```

```
 1.0+0.0im  1.0+0.0im  1.0+0.0im
```

`Base.BitArray` — Type

```
BitArray{N} <: AbstractArray{Bool, N}
```

Space-efficient `N`-dimensional boolean array, using just one bit for each boolean value.

`BitArray`s pack up to 64 values into every 8 bytes, resulting in an 8x space efficiency over `Array{Bool, N}` and allowing some operations to work on 64 values at once.

By default, Julia returns `BitArrays` from broadcasting operations that generate boolean elements (including dotted-comparisons like `.==`) as well as from the functions `trues` and `falses`.

> **Note**
>
> Due to its packed storage format, concurrent access to the elements of a `BitArray` where at least one of them is a write is not thread safe.

`Base.BitArray` — Method

```
BitArray(undef, dims::Integer...)
BitArray{N}(undef, dims::NTuple{N,Int})
```

Construct an undef `BitArray` with the given dimensions. Behaves identically to the `Array` constructor. See `undef`.

Examples

```
julia> BitArray(undef, 2, 2)
2×2 BitArray{2}:
 0  0
 0  0

julia> BitArray(undef, (3, 1))
3×1 BitArray{2}:
```

```
 0
 0
 0
```

Base.BitArray — Method

```
BitArray(itr)
```

Construct a `BitArray` generated by the given iterable object. The shape is inferred from the `itr` object.

Examples

```
julia> BitArray([1 0; 0 1])
2×2 BitArray{2}:
 1  0
 0  1

julia> BitArray(x+y == 3 for x = 1:2, y = 1:3)
2×3 BitArray{2}:
 0  1  0
 1  0  0

julia> BitArray(x+y == 3 for x = 1:2 for y = 1:3)
6-element BitArray{1}:
 0
 1
 0
 1
 0
 0
```

Base.trues — Function

```
trues(dims)
```

Create a `BitArray` with all values set to `true`.

Examples

```
julia> trues(2,3)
2×3 BitArray{2}:
 1  1  1
 1  1  1
```

`Base.falses` — Function

```
falses(dims)
```

Create a `BitArray` with all values set to `false`.

Examples

```
julia> falses(2,3)
2×3 BitArray{2}:
 0  0  0
 0  0  0
```

`Base.fill` — Function

```
fill(x, dims::Tuple)
fill(x, dims...)
```

Create an array filled with the value `x`. For example, `fill(1.0, (5,5))` returns a 5×5 array of floats, with each element initialized to `1.0`.

`dims` may be specified as either a tuple or a sequence of arguments. For example, the common idiom `fill(x)` creates a zero-dimensional array containing the single value `x`.

Examples

```
julia> fill(1.0, (2,3))
2×3 Array{Float64,2}:
 1.0  1.0  1.0
 1.0  1.0  1.0

julia> fill(42)
0-dimensional Array{Int64,0}:
```

```
42
```

If `x` is an object reference, all elements will refer to the same object:

```
julia> A = fill(zeros(2), 2);

julia> A[1][1] = 42; # modifies both A[1][1] and A[2][1]

julia> A
2-element Array{Array{Float64,1},1}:
 [42.0, 0.0]
 [42.0, 0.0]
```

`Base.fill!` — Function

```
fill!(A, x)
```

Fill array `A` with the value `x`. If `x` is an object reference, all elements will refer to the same object. `fill!(A, Foo())` will return `A` filled with the result of evaluating `Foo()` once.

Examples

```
julia> A = zeros(2,3)
2×3 Array{Float64,2}:
 0.0  0.0  0.0
 0.0  0.0  0.0

julia> fill!(A, 2.)
2×3 Array{Float64,2}:
 2.0  2.0  2.0
 2.0  2.0  2.0

julia> a = [1, 1, 1]; A = fill!(Vector{Vector{Int}}(undef, 3), a); a[1] = 2; A
3-element Array{Array{Int64,1},1}:
 [2, 1, 1]
 [2, 1, 1]
 [2, 1, 1]

julia> x = 0; f() = (global x += 1; x); fill!(Vector{Int}(undef, 3), f())
3-element Array{Int64,1}:
 1
```

```
 1
 1
```

`Base.similar` — Function

```
similar(array, [element_type=eltype(array)], [dims=size(array)])
```

Create an uninitialized mutable array with the given element type and size, based upon the given source array. The second and third arguments are both optional, defaulting to the given array's `eltype` and `size`. The dimensions may be specified either as a single tuple argument or as a series of integer arguments.

Custom AbstractArray subtypes may choose which specific array type is best-suited to return for the given element type and dimensionality. If they do not specialize this method, the default is an `Array{element_type}(undef, dims...)`.

For example, `similar(1:10, 1, 4)` returns an uninitialized `Array{Int,2}` since ranges are neither mutable nor support 2 dimensions:

```
julia> similar(1:10, 1, 4)
1×4 Array{Int64,2}:
 4419743872  4374413872  4419743888  0
```

Conversely, `similar(trues(10,10), 2)` returns an uninitialized `BitVector` with two elements since `BitArray`s are both mutable and can support 1-dimensional arrays:

```
julia> similar(trues(10,10), 2)
2-element BitArray{1}:
 0
 0
```

Since `BitArray`s can only store elements of type `Bool`, however, if you request a different element type it will create a regular `Array` instead:

```
julia> similar(falses(10), Float64, 2, 4)
2×4 Array{Float64,2}:
 2.18425e-314  2.18425e-314  2.18425e-314  2.18425e-314
 2.18425e-314  2.18425e-314  2.18425e-314  2.18425e-314
```

```
similar(storagetype, axes)
```

Create an uninitialized mutable array analogous to that specified by `storagetype`, but with `axes` specified by the last argument. `storagetype` might be a type or a function.

Examples:

```
similar(Array{Int}, axes(A))
```

creates an array that "acts like" an `Array{Int}` (and might indeed be backed by one), but which is indexed identically to `A`. If `A` has conventional indexing, this will be identical to `Array{Int}(undef, size(A))`, but if `A` has unconventional indexing then the indices of the result will match `A`.

```
similar(BitArray, (axes(A, 2),))
```

would create a 1-dimensional logical array whose indices match those of the columns of `A`.

Basic functions

`Base.ndims` — Function

```
ndims(A::AbstractArray) -> Integer
```

Return the number of dimensions of `A`.

Examples

```
julia> A = fill(1, (3,4,5));

julia> ndims(A)
3
```

`Base.size` — Function

```
size(A::AbstractArray, [dim])
```

Return a tuple containing the dimensions of `A`. Optionally you can specify a dimension to just get the length of that dimension.

Note that `size` may not be defined for arrays with non-standard indices, in which case `axes` may be useful. See the manual chapter on arrays with custom indices.

Examples

```
julia> A = fill(1, (2,3,4));

julia> size(A)
(2, 3, 4)

julia> size(A, 2)
3
```

`Base.axes` — Method

```
axes(A)
```

Return the tuple of valid indices for array `A`.

Examples

```
julia> A = fill(1, (5,6,7));

julia> axes(A)
(Base.OneTo(5), Base.OneTo(6), Base.OneTo(7))
```

`Base.axes` — Method

```
axes(A, d)
```

Return the valid range of indices for array `A` along dimension `d`.

See also `size`, and the manual chapter on arrays with custom indices.

Examples

```
julia> A = fill(1, (5,6,7));

julia> axes(A, 2)
Base.OneTo(6)
```

`Base.length` — Method

```
length(A::AbstractArray)
```

Return the number of elements in the array, defaults to `prod(size(A))`.

Examples

```
julia> length([1, 2, 3, 4])
4

julia> length([1 2; 3 4])
4
```

`Base.eachindex` — Function

```
eachindex(A...)
```

Create an iterable object for visiting each index of an `AbstractArray A` in an efficient manner. For array types that have opted into fast linear indexing (like `Array`), this is simply the range `1:length(A)`. For other array types, return a specialized Cartesian range to efficiently index into the array with indices specified for every dimension. For other iterables, including strings and dictionaries, return an iterator object supporting arbitrary index types (e.g. unevenly spaced or non-integer indices).

If you supply more than one `AbstractArray` argument, `eachindex` will create an iterable object that is fast for all arguments (a `UnitRange` if all inputs have fast linear indexing, a `CartesianIndices` otherwise). If the arrays have different sizes and/or dimensionalities, a

DimensionMismatch exception will be thrown.

Examples

```
julia> A = [1 2; 3 4];

julia> for i in eachindex(A) # linear indexing
           println(i)
       end
1
2
3
4

julia> for i in eachindex(view(A, 1:2, 1:1)) # Cartesian indexing
           println(i)
       end
CartesianIndex(1, 1)
CartesianIndex(2, 1)
```

`Base.IndexStyle` — Type

```
IndexStyle(A)
IndexStyle(typeof(A))
```

`IndexStyle` specifies the "native indexing style" for array `A`. When you define a new `AbstractArray` type, you can choose to implement either linear indexing (with `IndexLinear`) or cartesian indexing. If you decide to only implement linear indexing, then you must set this trait for your array type:

```
Base.IndexStyle(::Type{<:MyArray}) = IndexLinear()
```

The default is `IndexCartesian()`.

Julia's internal indexing machinery will automatically (and invisibly) recompute all indexing operations into the preferred style. This allows users to access elements of your array using any indexing style, even when explicit methods have not been provided.

If you define both styles of indexing for your `AbstractArray`, this trait can be used to select the most performant indexing style. Some methods check this trait on their inputs, and dispatch to different algorithms depending on the most efficient access pattern. In particular, `eachindex` creates an iterator whose type depends on the setting of this trait.

`Base.IndexLinear` — Type

```
IndexLinear()
```

Subtype of `IndexStyle` used to describe arrays which are optimally indexed by one linear index.

A linear indexing style uses one integer index to describe the position in the array (even if it's a multidimensional array) and column-major ordering is used to efficiently access the elements. This means that requesting `eachindex` from an array that is `IndexLinear` will return a simple one-dimensional range, even if it is multidimensional.

A custom array that reports its `IndexStyle` as `IndexLinear` only needs to implement indexing (and indexed assignment) with a single `Int` index; all other indexing expressions — including multidimensional accesses — will be recomputed to the linear index. For example, if `A` were a `2×3` custom matrix with linear indexing, and we referenced `A[1, 3]`, this would be recomputed to the equivalent linear index and call `A[5]` since `2*1 + 3 = 5`.

See also `IndexCartesian`.

`Base.IndexCartesian` — Type

```
IndexCartesian()
```

Subtype of `IndexStyle` used to describe arrays which are optimally indexed by a Cartesian index. This is the default for new custom `AbstractArray` subtypes.

A Cartesian indexing style uses multiple integer indices to describe the position in a multidimensional array, with exactly one index per dimension. This means that requesting `eachindex` from an array that is `IndexCartesian` will return a range of `CartesianIndices`.

A `N`-dimensional custom array that reports its `IndexStyle` as `IndexCartesian` needs to implement indexing (and indexed assignment) with exactly `N` `Int` indices; all other indexing expressions — including linear indexing — will be recomputed to the equivalent Cartesian location. For example, if `A` were a `2×3` custom matrix with cartesian indexing, and we referenced `A[5]`, this would be recomputed to the equivalent Cartesian index and call `A[1, 3]` since `5 = 2*1 + 3`.

It is significantly more expensive to compute Cartesian indices from a linear index than it is to go the other way. The former operation requires division — a very costly operation — whereas the latter only uses multiplication and addition and is essentially free. This asymmetry means it is far more costly to use linear indexing with an `IndexCartesian` array than it is to use Cartesian indexing with an `IndexLinear` array.

See also `IndexLinear`.

`Base.conj!` — Function

```
conj!(A)
```

Transform an array to its complex conjugate in-place.

See also `conj`.

Examples

```
julia> A = [1+im 2-im; 2+2im 3+im]
2×2 Array{Complex{Int64},2}:
 1+1im  2-1im
 2+2im  3+1im

julia> conj!(A);

julia> A
2×2 Array{Complex{Int64},2}:
 1-1im  2+1im
 2-2im  3-1im
```

`Base.stride` — Function

```
stride(A, k::Integer)
```

Return the distance in memory (in number of elements) between adjacent elements in dimension `k`.

Examples

```
julia> A = fill(1, (3,4,5));

julia> stride(A,2)
3

julia> stride(A,3)
12
```

`Base.strides` — Function

```
strides(A)
```

Return a tuple of the memory strides in each dimension.

Examples

```
julia> A = fill(1, (3,4,5));

julia> strides(A)
(1, 3, 12)
```

Broadcast and vectorization

See also the dot syntax for vectorizing functions; for example, `f.(args...)` implicitly calls `broadcast(f, args...)`. Rather than relying on "vectorized" methods of functions like `sin` to operate on arrays, you should use `sin.(a)` to vectorize via `broadcast`.

`Base.Broadcast.broadcast` — Function

```
broadcast(f, As...)
```

Broadcast the function `f` over the arrays, tuples, collections, `Ref`s and/or scalars `As`.

Broadcasting applies the function `f` over the elements of the container arguments and the scalars themselves in `As`. Singleton and missing dimensions are expanded to match the extents of the other arguments by virtually repeating the value. By default, only a limited number of types are considered scalars, including `Number`s, `String`s, `Symbol`s, `Type`s, `Function`s and some common singletons like `missing` and `nothing`. All other arguments are iterated over or indexed into elementwise.

The resulting container type is established by the following rules:

- If all the arguments are scalars or zero-dimensional arrays, it returns an unwrapped scalar.
- If at least one argument is a tuple and all others are scalars or zero-dimensional arrays, it returns a tuple.
- All other combinations of arguments default to returning an `Array`, but custom container types can define their own implementation and promotion-like rules to customize the result when they appear as arguments.

A special syntax exists for broadcasting: `f.(args...)` is equivalent to `broadcast(f, args...)`, and nested `f.(g.(args...))` calls are fused into a single broadcast loop.

Examples

```
julia> A = [1, 2, 3, 4, 5]
5-element Array{Int64,1}:
 1
 2
 3
 4
 5

julia> B = [1 2; 3 4; 5 6; 7 8; 9 10]
5×2 Array{Int64,2}:
 1   2
 3   4
 5   6
 7   8
 9  10

julia> broadcast(+, A, B)
5×2 Array{Int64,2}:
  2   3
  5   6
  8   9
 11  12
 14  15

julia> parse.(Int, ["1", "2"])
2-element Array{Int64,1}:
 1
```

```
 2

julia> abs.((1, -2))
(1, 2)

julia> broadcast(+, 1.0, (0, -2.0))
(1.0, -1.0)

julia> (+).([[0,2], [1,3]], Ref{Vector{Int}}([1,-1]))
2-element Array{Array{Int64,1},1}:
 [1, 1]
 [2, 2]

julia> string.(("one","two","three","four"), ": ", 1:4)
4-element Array{String,1}:
 "one: 1"
 "two: 2"
 "three: 3"
 "four: 4"
```

`Base.Broadcast.broadcast!` — Function

```
broadcast!(f, dest, As...)
```

Like `broadcast`, but store the result of `broadcast(f, As...)` in the `dest` array. Note that `dest` is only used to store the result, and does not supply arguments to `f` unless it is also listed in the `As`, as in `broadcast!(f, A, A, B)` to perform `A[:] = broadcast(f, A, B)`.

Examples

```
julia> A = [1.0; 0.0]; B = [0.0; 0.0];

julia> broadcast!(+, B, A, (0, -2.0));

julia> B
2-element Array{Float64,1}:
  1.0
 -2.0

julia> A
2-element Array{Float64,1}:
 1.0
```

```
 0.0

julia> broadcast!(+, A, A, (0, -2.0));

julia> A
2-element Array{Float64,1}:
  1.0
 -2.0
```

`Base.Broadcast.@__dot__` — Macro

```
@. expr
```

Convert every function call or operator in `expr` into a "dot call" (e.g. convert `f(x)` to `f.(x)`), and convert every assignment in `expr` to a "dot assignment" (e.g. convert += to `.+=`).

If you want to *avoid* adding dots for selected function calls in `expr`, splice those function calls in with `$`. For example, `@. sqrt(abs($sort(x)))` is equivalent to `sqrt.(abs.(sort(x)))` (no dot for `sort`).

(`@.` is equivalent to a call to `@__dot__`.)

Examples

```
julia> x = 1.0:3.0; y = similar(x);

julia> @. y = x + 3 * sin(x)
3-element Array{Float64,1}:
 3.5244129544236893
 4.727892280477045
 3.4233600241796016
```

For specializing broadcast on custom types, see

`Base.Broadcast.BroadcastStyle` — Type

`BroadcastStyle` is an abstract type and trait-function used to determine behavior of objects under broadcasting. `BroadcastStyle(typeof(x))` returns the style associated with `x`. To customize the broadcasting behavior of a type, one can declare a style by defining a type/method pair

```
struct MyContainerStyle <: BroadcastStyle end
Base.BroadcastStyle(::Type{<:MyContainer}) = MyContainerStyle()
```

One then writes method(s) (at least `similar`) operating on `Broadcasted{MyContainerStyle}`. There are also several pre-defined subtypes of `BroadcastStyle` that you may be able to leverage; see the Interfaces chapter for more information.

`Base.Broadcast.AbstractArrayStyle` — Type

`Broadcast.AbstractArrayStyle{N} <: BroadcastStyle` is the abstract supertype for any style associated with an `AbstractArray` type. The `N` parameter is the dimensionality, which can be handy for AbstractArray types that only support specific dimensionalities:

```
struct SparseMatrixStyle <: Broadcast.AbstractArrayStyle{2} end
Base.BroadcastStyle(::Type{<:SparseMatrixCSC}) = SparseMatrixStyle()
```

For `AbstractArray` types that support arbitrary dimensionality, `N` can be set to `Any`:

```
struct MyArrayStyle <: Broadcast.AbstractArrayStyle{Any} end
Base.BroadcastStyle(::Type{<:MyArray}) = MyArrayStyle()
```

In cases where you want to be able to mix multiple `AbstractArrayStyle`s and keep track of dimensionality, your style needs to support a `Val` constructor:

```
struct MyArrayStyleDim{N} <: Broadcast.AbstractArrayStyle{N} end
(::Type{<:MyArrayStyleDim})(::Val{N}) where N = MyArrayStyleDim{N}()
```

Note that if two or more `AbstractArrayStyle` subtypes conflict, broadcasting machinery will fall back to producing `Array`s. If this is undesirable, you may need to define binary `BroadcastStyle` rules to control the output type.

See also `Broadcast.DefaultArrayStyle`.

`Base.Broadcast.ArrayStyle` — Type

`Broadcast.ArrayStyle{MyArrayType}()` is a `BroadcastStyle` indicating that an object behaves as an array for broadcasting. It presents a simple way to construct `Broadcast.AbstractArrayStyle`s for specific `AbstractArray` container types. Broadcast styles created this way lose track of dimensionality; if keeping track is important for your type, you should create your own custom `Broadcast.AbstractArrayStyle`.

`Base.Broadcast.DefaultArrayStyle` — Type

`Broadcast.DefaultArrayStyle{N}()` is a `BroadcastStyle` indicating that an object behaves as an `N`-dimensional array for broadcasting. Specifically, `DefaultArrayStyle` is used for any `AbstractArray` type that hasn't defined a specialized style, and in the absence of overrides from other `broadcast` arguments the resulting output type is `Array`. When there are multiple inputs to `broadcast`, `DefaultArrayStyle` "loses" to any other `Broadcast.ArrayStyle`.

`Base.Broadcast.broadcastable` — Function

```
Broadcast.broadcastable(x)
```

Return either `x` or an object like `x` such that it supports `axes`, indexing, and its type supports `ndims`.

If `x` supports iteration, the returned value should have the same `axes` and indexing behaviors as `collect(x)`.

If `x` is not an `AbstractArray` but it supports `axes`, indexing, and its type supports `ndims`, then `broadcastable(::typeof(x))` may be implemented to just return itself. Further, if `x` defines its own `BroadcastStyle`, then it must define its `broadcastable` method to return itself for the custom style to have any effect.

Examples

```
julia> Broadcast.broadcastable([1,2,3]) # like `identity` since arrays already
3-element Array{Int64,1}:
 1
 2
 3

julia> Broadcast.broadcastable(Int) # Types don't support axes, indexing, or it
Base.RefValue{Type{Int64}}(Int64)
```

```
julia> Broadcast.broadcastable("hello") # Strings break convention of matching
Base.RefValue{String}("hello")
```

`Base.Broadcast.combine_axes` — Function

```
combine_axes(As...) -> Tuple
```

Determine the result axes for broadcasting across all values in `As`.

```
julia> Broadcast.combine_axes([1], [1 2; 3 4; 5 6])
(Base.OneTo(3), Base.OneTo(2))

julia> Broadcast.combine_axes(1, 1, 1)
()
```

`Base.Broadcast.combine_styles` — Function

```
combine_styles(cs...) -> BroadcastStyle
```

Decides which `BroadcastStyle` to use for any number of value arguments. Uses `BroadcastStyle` to get the style for each argument, and uses `result_style` to combine styles.

Examples

```
julia> Broadcast.combine_styles([1], [1 2; 3 4])
Base.Broadcast.DefaultArrayStyle{2}()
```

`Base.Broadcast.result_style` — Function

```
result_style(s1::BroadcastStyle[, s2::BroadcastStyle]) -> BroadcastStyle
```

Takes one or two `BroadcastStyle`s and combines them using `BroadcastStyle` to determine a common `BroadcastStyle`.

Examples

```
julia> Broadcast.result_style(Broadcast.DefaultArrayStyle{0}(), Broadcast.Defau
Base.Broadcast.DefaultArrayStyle{3}()

julia> Broadcast.result_style(Broadcast.Unknown(), Broadcast.DefaultArrayStyle{
Base.Broadcast.DefaultArrayStyle{1}()
```

Indexing and assignment

`Base.getindex` — Method

```
getindex(A, inds...)
```

Return a subset of array `A` as specified by `inds`, where each `ind` may be an `Int`, an `AbstractRange`, or a `Vector`. See the manual section on array indexing for details.

Examples

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

julia> getindex(A, 1)
1

julia> getindex(A, [2, 1])
2-element Array{Int64,1}:
 3
 1

julia> getindex(A, 2:4)
3-element Array{Int64,1}:
 3
 2
 4
```

`Base.setindex!` — Method

```
setindex!(A, X, inds...)
A[inds...] = X
```

Store values from array `X` within some subset of `A` as specified by `inds`. The syntax `A[inds...] = X` is equivalent to `setindex!(A, X, inds...)`.

Examples

```
julia> A = zeros(2,2);

julia> setindex!(A, [10, 20], [1, 2]);

julia> A[[3, 4]] = [30, 40];

julia> A
2×2 Array{Float64,2}:
 10.0  30.0
 20.0  40.0
```

`Base.copyto!` — Method

```
copyto!(dest, Rdest::CartesianIndices, src, Rsrc::CartesianIndices) -> dest
```

Copy the block of `src` in the range of `Rsrc` to the block of `dest` in the range of `Rdest`. The sizes of the two regions must match.

`Base.isassigned` — Function

```
isassigned(array, i) -> Bool
```

Test whether the given array has a value associated with index `i`. Return `false` if the index is out of bounds, or has an undefined reference.

Examples

```
julia> isassigned(rand(3, 3), 5)
true
```

```
julia> isassigned(rand(3, 3), 3 * 3 + 1)
false

julia> mutable struct Foo end

julia> v = similar(rand(3), Foo)
3-element Array{Foo,1}:
 #undef
 #undef
 #undef

julia> isassigned(v, 1)
false
```

`Base.Colon` — Type

```
Colon()
```

Colons (:) are used to signify indexing entire objects or dimensions at once.

Very few operations are defined on Colons directly; instead they are converted by `to_indices` to an internal vector type (`Base.Slice`) to represent the collection of indices they span before being used.

The singleton instance of `Colon` is also a function used to construct ranges; see `:`.

`Base.IteratorsMD.CartesianIndex` — Type

```
CartesianIndex(i, j, k...)   -> I
CartesianIndex((i, j, k...)) -> I
```

Create a multidimensional index `I`, which can be used for indexing a multidimensional array `A`. In particular, `A[I]` is equivalent to `A[i,j,k...]`. One can freely mix integer and `CartesianIndex` indices; for example, `A[Ipre, i, Ipost]` (where `Ipre` and `Ipost` are `CartesianIndex` indices and `i` is an `Int`) can be a useful expression when writing algorithms that work along a single dimension of an array of arbitrary dimensionality.

A `CartesianIndex` is sometimes produced by `eachindex`, and always when iterating with an explicit `CartesianIndices`.

Examples

```
julia> A = reshape(Vector(1:16), (2, 2, 2, 2))
2×2×2×2 Array{Int64,4}:
[:, :, 1, 1] =
 1  3
 2  4

[:, :, 2, 1] =
 5  7
 6  8

[:, :, 1, 2] =
  9  11
 10  12

[:, :, 2, 2] =
 13  15
 14  16

julia> A[CartesianIndex((1, 1, 1, 1))]
1

julia> A[CartesianIndex((1, 1, 1, 2))]
9

julia> A[CartesianIndex((1, 1, 2, 1))]
5
```

`Base.IteratorsMD.CartesianIndices` — Type

```
CartesianIndices(sz::Dims) -> R
CartesianIndices((istart:istop, jstart:jstop, ...)) -> R
```

Define a region `R` spanning a multidimensional rectangular range of integer indices. These are most commonly encountered in the context of iteration, where `for I in R ... end` will return `CartesianIndex` indices `I` equivalent to the nested loops

```
for j = jstart:jstop
```

```
    for i = istart:istop
        ...
    end
end
```

Consequently these can be useful for writing algorithms that work in arbitrary dimensions.

```
CartesianIndices(A::AbstractArray) -> R
```

As a convenience, constructing a `CartesianIndices` from an array makes a range of its indices.

Examples

```
julia> foreach(println, CartesianIndices((2, 2, 2)))
CartesianIndex(1, 1, 1)
CartesianIndex(2, 1, 1)
CartesianIndex(1, 2, 1)
CartesianIndex(2, 2, 1)
CartesianIndex(1, 1, 2)
CartesianIndex(2, 1, 2)
CartesianIndex(1, 2, 2)
CartesianIndex(2, 2, 2)

julia> CartesianIndices(fill(1, (2,3)))
2×3 CartesianIndices{2,Tuple{Base.OneTo{Int64},Base.OneTo{Int64}}}:
 CartesianIndex(1, 1)  CartesianIndex(1, 2)  CartesianIndex(1, 3)
 CartesianIndex(2, 1)  CartesianIndex(2, 2)  CartesianIndex(2, 3)
```

Conversion between linear and cartesian indices

Linear index to cartesian index conversion exploits the fact that a `CartesianIndices` is an `AbstractArray` and can be indexed linearly:

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

julia> cartesian[4]
CartesianIndex(1, 2)
```

Broadcasting

`CartesianIndices` support broadcasting arithmetic (+ and -) with a `CartesianIndex`.

> **Julia 1.1**
>
> Broadcasting of CartesianIndices requires at least Julia 1.1.

```
julia> CIs = CartesianIndices((2:3, 5:6))
2×2 CartesianIndices{2,Tuple{UnitRange{Int64},UnitRange{Int64}}}:
 CartesianIndex(2, 5)  CartesianIndex(2, 6)
 CartesianIndex(3, 5)  CartesianIndex(3, 6)

julia> CI = CartesianIndex(3, 4)
CartesianIndex(3, 4)

julia> CIs .+ CI
2×2 CartesianIndices{2,Tuple{UnitRange{Int64},UnitRange{Int64}}}:
 CartesianIndex(5, 9)  CartesianIndex(5, 10)
 CartesianIndex(6, 9)  CartesianIndex(6, 10)
```

For cartesian to linear index conversion, see `LinearIndices`.

`Base.Dims` — Type

```
Dims{N}
```

An `NTuple` of `N` `Int`s used to represent the dimensions of an `AbstractArray`.

`Base.LinearIndices` — Type

```
LinearIndices(A::AbstractArray)
```

Return a `LinearIndices` array with the same shape and `axes` as `A`, holding the linear index of each entry in `A`. Indexing this array with cartesian indices allows mapping them to linear indices.

For arrays with conventional indexing (indices start at 1), or any multidimensional array, linear indices range from 1 to `length(A)`. However, for `AbstractVector`s linear indices are `axes(A, 1)`, and therefore do not start at 1 for vectors with unconventional indexing.

Calling this function is the "safe" way to write algorithms that exploit linear indexing.

Examples

```
julia> A = fill(1, (5,6,7));

julia> b = LinearIndices(A);

julia> extrema(b)
(1, 210)
```

```
LinearIndices(inds::CartesianIndices) -> R
LinearIndices(sz::Dims) -> R
LinearIndices((istart:istop, jstart:jstop, ...)) -> R
```

Return a `LinearIndices` array with the specified shape or `axes`.

Example

The main purpose of this constructor is intuitive conversion from cartesian to linear indexing:

```
julia> linear = LinearIndices((1:3, 1:2))
3×2 LinearIndices{2,Tuple{UnitRange{Int64},UnitRange{Int64}}}:
 1  4
 2  5
 3  6

julia> linear[1,2]
4
```

`Base.to_indices` — Function

```
to_indices(A, I::Tuple)
```

Convert the tuple `I` to a tuple of indices for use in indexing into array `A`.

The returned tuple must only contain either `Int`s or `AbstractArray`s of scalar indices that are supported by array `A`. It will error upon encountering a novel index type that it does not know how to process.

For simple index types, it defers to the unexported `Base.to_index(A, i)` to process each index `i`. While this internal function is not intended to be called directly, `Base.to_index` may be extended by custom array or index types to provide custom indexing behaviors.

More complicated index types may require more context about the dimension into which they index. To support those cases, `to_indices(A, I)` calls `to_indices(A, axes(A), I)`, which then recursively walks through both the given tuple of indices and the dimensional indices of `A` in tandem. As such, not all index types are guaranteed to propagate to `Base.to_index`.

`Base.checkbounds` — Function

```
checkbounds(Bool, A, I...)
```

Return `true` if the specified indices `I` are in bounds for the given array `A`. Subtypes of `AbstractArray` should specialize this method if they need to provide custom bounds checking behaviors; however, in many cases one can rely on `A`'s indices and `checkindex`.

See also `checkindex`.

Examples

```
julia> A = rand(3, 3);

julia> checkbounds(Bool, A, 2)
true

julia> checkbounds(Bool, A, 3, 4)
false

julia> checkbounds(Bool, A, 1:3)
true

julia> checkbounds(Bool, A, 1:3, 2:4)
false
```

```
checkbounds(A, I...)
```

Throw an error if the specified indices `I` are not in bounds for the given array `A`.

`Base.checkindex` — Function

```
checkindex(Bool, inds::AbstractUnitRange, index)
```

Return `true` if the given `index` is within the bounds of `inds`. Custom types that would like to behave as indices for all arrays can extend this method in order to provide a specialized bounds checking implementation.

Examples

```
julia> checkindex(Bool, 1:20, 8)
true

julia> checkindex(Bool, 1:20, 21)
false
```

Views (SubArrays and other view types)

A "view" is a data structure that acts like an array (it is a subtype of `AbstractArray`), but the underlying data is actually part of another array.

For example, if `x` is an array and `v = @view x[1:10]`, then `v` acts like a 10-element array, but its data is actually accessing the first 10 elements of `x`. Writing to a view, e.g. `v[3] = 2`, writes directly to the underlying array `x` (in this case modifying `x[3]`).

Slicing operations like `x[1:10]` create a copy by default in Julia. `@view x[1:10]` changes it to make a view. The `@views` macro can be used on a whole block of code (e.g. `@views function foo() .... end` or `@views begin ... end`) to change all the slicing operations in that block to use views. Sometimes making a copy of the data is faster and sometimes using a view is faster, as described in the performance tips.

`Base.view` — Function

```
view(A, inds...)
```

Like `getindex`, but returns a view into the parent array `A` with the given indices instead of making a copy. Calling `getindex` or `setindex!` on the returned `SubArray` computes the indices to the parent array on the fly without checking bounds.

Examples

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

julia> b = view(A, :, 1)
2-element view(::Array{Int64,2}, :, 1) with eltype Int64:
 1
 3

julia> fill!(b, 0)
2-element view(::Array{Int64,2}, :, 1) with eltype Int64:
 0
 0

julia> A # Note A has changed even though we modified b
2×2 Array{Int64,2}:
 0  2
 0  4
```

`Base.@view` — Macro

```
@view A[inds...]
```

Creates a `SubArray` from an indexing expression. This can only be applied directly to a reference expression (e.g. `@view A[1,2:end]`), and should *not* be used as the target of an assignment (e.g. `@view(A[1,2:end]) = ...`). See also `@views` to switch an entire block of code to use views for slicing.

> **Julia 1.5**
>
> Using `begin` in an indexing expression to refer to the first index requires at least Julia 1.5.

Examples

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

julia> b = @view A[:, 1]
2-element view(::Array{Int64,2}, :, 1) with eltype Int64:
 1
 3

julia> fill!(b, 0)
2-element view(::Array{Int64,2}, :, 1) with eltype Int64:
 0
 0

julia> A
2×2 Array{Int64,2}:
 0  2
 0  4
```

`Base.@views` — Macro

```
@views expression
```

Convert every array-slicing operation in the given expression (which may be a `begin`/`end` block, loop, function, etc.) to return a view. Scalar indices, non-array types, and explicit `getindex` calls (as opposed to `array[...]`) are unaffected.

> **Note**
>
> The `@views` macro only affects `array[...]` expressions that appear explicitly in the given `expression`, not array slicing that occurs in functions called by that code.

> **Julia 1.5**
>
> Using `begin` in an indexing expression to refer to the first index requires at least Julia 1.5.

Examples

```
julia> A = zeros(3, 3);

julia> @views for row in 1:3
           b = A[row, :]
           b[:] .= row
       end

julia> A
3×3 Array{Float64,2}:
 1.0  1.0  1.0
 2.0  2.0  2.0
 3.0  3.0  3.0
```

`Base.parent` — Function

```
parent(A)
```

Return the underlying "parent array". This parent array of objects of types `SubArray`, `ReshapedArray` or `LinearAlgebra.Transpose` is what was passed as an argument to `view`, `reshape`, `transpose`, etc. during object creation. If the input is not a wrapped object, return the input itself.

Examples

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

julia> V = view(A, 1:2, :)
2×2 view(::Array{Int64,2}, 1:2, :) with eltype Int64:
 1  2
 3  4

julia> parent(V)
2×2 Array{Int64,2}:
 1  2
 3  4
```

`Base.parentindices` — Function

```
parentindices(A)
```

Return the indices in the `parent` which correspond to the array view `A`.

Examples

```
julia> A = [1 2; 3 4];

julia> V = view(A, 1, :)
2-element view(::Array{Int64,2}, 1, :) with eltype Int64:
 1
 2

julia> parentindices(V)
(1, Base.Slice(Base.OneTo(2)))
```

`Base.selectdim` — Function

```
selectdim(A, d::Integer, i)
```

Return a view of all the data of `A` where the index for dimension `d` equals `i`.

Equivalent to `view(A,:,:,...,i,:,:,...)` where `i` is in position `d`.

Examples

```
julia> A = [1 2 3 4; 5 6 7 8]
2×4 Array{Int64,2}:
 1  2  3  4
 5  6  7  8

julia> selectdim(A, 2, 3)
2-element view(::Array{Int64,2}, :, 3) with eltype Int64:
 3
 7
```

`Base.reinterpret` — Function

```
reinterpret(type, A)
```

Change the type-interpretation of a block of memory. For arrays, this constructs a view of the array with the same binary data as the given array, but with the specified element type. For example, `reinterpret(Float32, UInt32(7))` interprets the 4 bytes corresponding to `UInt32(7)` as a `Float32`.

Examples

```
julia> reinterpret(Float32, UInt32(7))
1.0f-44

julia> reinterpret(Float32, UInt32[1 2 3 4 5])
1×5 reinterpret(Float32, ::Array{UInt32,2}):
 1.0f-45  3.0f-45  4.0f-45  6.0f-45  7.0f-45
```

`Base.reshape` — Function

```
reshape(A, dims...) -> AbstractArray
reshape(A, dims) -> AbstractArray
```

Return an array with the same data as `A`, but with different dimension sizes or number of dimensions. The two arrays share the same underlying data, so that the result is mutable if and only if `A` is mutable, and setting elements of one alters the values of the other.

The new dimensions may be specified either as a list of arguments or as a shape tuple. At most one dimension may be specified with a `:`, in which case its length is computed such that its product with all the specified dimensions is equal to the length of the original array `A`. The total number of elements must not change.

Examples

```
julia> A = Vector(1:16)
16-element Array{Int64,1}:
  1
  2
```

```
  3
  4
  5
  6
  7
  8
  9
 10
 11
 12
 13
 14
 15
 16

julia> reshape(A, (4, 4))
4×4 Array{Int64,2}:
 1  5   9  13
 2  6  10  14
 3  7  11  15
 4  8  12  16

julia> reshape(A, 2, :)
2×8 Array{Int64,2}:
 1  3  5  7   9  11  13  15
 2  4  6  8  10  12  14  16

julia> reshape(1:6, 2, 3)
2×3 reshape(::UnitRange{Int64}, 2, 3) with eltype Int64:
 1  3  5
 2  4  6
```

`Base.dropdims` — Function

```
dropdims(A; dims)
```

Remove the dimensions specified by `dims` from array `A`. Elements of `dims` must be unique and within the range `1:ndims(A)`. `size(A,i)` must equal 1 for all `i` in `dims`.

Examples

```
julia> a = reshape(Vector(1:4),(2,2,1,1))
```

```
2×2×1×1 Array{Int64,4}:
[:, :, 1, 1] =
 1  3
 2  4

julia> dropdims(a; dims=3)
2×2×1 Array{Int64,3}:
[:, :, 1] =
 1  3
 2  4
```

`Base.vec` — Function

```
vec(a::AbstractArray) -> AbstractVector
```

Reshape the array `a` as a one-dimensional column vector. Return `a` if it is already an `AbstractVector`. The resulting array shares the same underlying data as `a`, so it will only be mutable if `a` is mutable, in which case modifying one will also modify the other.

Examples

```
julia> a = [1 2 3; 4 5 6]
2×3 Array{Int64,2}:
 1  2  3
 4  5  6

julia> vec(a)
6-element Array{Int64,1}:
 1
 4
 2
 5
 3
 6

julia> vec(1:3)
1:3
```

See also `reshape`.

Concatenation and permutation

`Base.cat` — Function

```
cat(A...; dims=dims)
```

Concatenate the input arrays along the specified dimensions in the iterable `dims`. For dimensions not in `dims`, all input arrays should have the same size, which will also be the size of the output array along that dimension. For dimensions in `dims`, the size of the output array is the sum of the sizes of the input arrays along that dimension. If `dims` is a single number, the different arrays are tightly stacked along that dimension. If `dims` is an iterable containing several dimensions, this allows one to construct block diagonal matrices and their higher-dimensional analogues by simultaneously increasing several dimensions for every new input array and putting zero blocks elsewhere. For example, `cat(matrices...; dims=(1,2))` builds a block diagonal matrix, i.e. a block matrix with `matrices[1]`, `matrices[2]`, ... as diagonal blocks and matching zero blocks away from the diagonal.

`Base.vcat` — Function

```
vcat(A...)
```

Concatenate along dimension 1.

Examples

```
julia> a = [1 2 3 4 5]
1×5 Array{Int64,2}:
 1  2  3  4  5

julia> b = [6 7 8 9 10; 11 12 13 14 15]
2×5 Array{Int64,2}:
  6   7   8   9  10
 11  12  13  14  15

julia> vcat(a,b)
3×5 Array{Int64,2}:
  1   2   3   4   5
  6   7   8   9  10
 11  12  13  14  15
```

```
julia> c = ([1 2 3], [4 5 6])
([1 2 3], [4 5 6])

julia> vcat(c...)
2×3 Array{Int64,2}:
 1  2  3
 4  5  6
```

`Base.hcat` — Function

```
hcat(A...)
```

Concatenate along dimension 2.

Examples

```
julia> a = [1; 2; 3; 4; 5]
5-element Array{Int64,1}:
 1
 2
 3
 4
 5

julia> b = [6 7; 8 9; 10 11; 12 13; 14 15]
5×2 Array{Int64,2}:
  6   7
  8   9
 10  11
 12  13
 14  15

julia> hcat(a,b)
5×3 Array{Int64,2}:
 1   6   7
 2   8   9
 3  10  11
 4  12  13
 5  14  15

julia> c = ([1; 2; 3], [4; 5; 6])
```

```
([1, 2, 3], [4, 5, 6])

julia> hcat(c...)
3×2 Array{Int64,2}:
 1  4
 2  5
 3  6

julia> x = Matrix(undef, 3, 0)  # x = [] would have created an Array{Any, 1}, b
3×0 Array{Any,2}

julia> hcat(x, [1; 2; 3])
3×1 Array{Any,2}:
 1
 2
 3
```

`Base.hvcat` — Function

```
hvcat(rows::Tuple{Vararg{Int}}, values...)
```

Horizontal and vertical concatenation in one call. This function is called for block matrix syntax. The first argument specifies the number of arguments to concatenate in each block row.

Examples

```
julia> a, b, c, d, e, f = 1, 2, 3, 4, 5, 6
(1, 2, 3, 4, 5, 6)

julia> [a b c; d e f]
2×3 Array{Int64,2}:
 1  2  3
 4  5  6

julia> hcat((3,3), a,b,c,d,e,f)
2×3 Array{Int64,2}:
 1  2  3
 4  5  6

julia> [a b;c d; e f]
3×2 Array{Int64,2}:
 1  2
```

```
 3  4
 5  6

julia> hvcat((2,2,2), a,b,c,d,e,f)
3×2 Array{Int64,2}:
 1  2
 3  4
 5  6
```

If the first argument is a single integer `n`, then all block rows are assumed to have `n` block columns.

`Base.vect` — Function

```
vect(X...)
```

Create a `Vector` with element type computed from the `promote_typeof` of the argument, containing the argument list.

Examples

```
julia> a = Base.vect(UInt8(1), 2.5, 1//2)
3-element Array{Float64,1}:
 1.0
 2.5
 0.5
```

`Base.circshift` — Function

```
circshift(A, shifts)
```

Circularly shift, i.e. rotate, the data in an array. The second argument is a tuple or vector giving the amount to shift in each dimension, or an integer to shift only in the first dimension.

Examples

```
julia> b = reshape(Vector(1:16), (4,4))
4×4 Array{Int64,2}:
 1  5   9  13
```

```
 2  6  10  14
 3  7  11  15
 4  8  12  16

julia> circshift(b, (0,2))
4×4 Array{Int64,2}:
  9  13  1  5
 10  14  2  6
 11  15  3  7
 12  16  4  8

julia> circshift(b, (-1,0))
4×4 Array{Int64,2}:
 2  6  10  14
 3  7  11  15
 4  8  12  16
 1  5   9  13

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

julia> circshift(a, 1)
5-element BitArray{1}:
 1
 1
 1
 0
 0

julia> circshift(a, -1)
5-element BitArray{1}:
 1
 0
 0
 1
 1
```

See also `circshift!`.

`Base.circshift!` — Function

```
circshift!(dest, src, shifts)
```

Circularly shift, i.e. rotate, the data in `src`, storing the result in `dest`. `shifts` specifies the amount to shift in each dimension.

The `dest` array must be distinct from the `src` array (they cannot alias each other).

See also `circshift`.

`Base.circcopy!` — Function

```
circcopy!(dest, src)
```

Copy `src` to `dest`, indexing each dimension modulo its length. `src` and `dest` must have the same size, but can be offset in their indices; any offset results in a (circular) wraparound. If the arrays have overlapping indices, then on the domain of the overlap `dest` agrees with `src`.

Examples

```
julia> src = reshape(Vector(1:16), (4,4))
4×4 Array{Int64,2}:
 1  5   9  13
 2  6  10  14
 3  7  11  15
 4  8  12  16

julia> dest = OffsetArray{Int}(undef, (0:3,2:5))

julia> circcopy!(dest, src)
OffsetArrays.OffsetArray{Int64,2,Array{Int64,2}} with indices 0:3×2:5:
 8  12  16  4
 5   9  13  1
 6  10  14  2
 7  11  15  3

julia> dest[1:3,2:4] == src[1:3,2:4]
true
```

`Base.findall` — Method

```
findall(A)
```

Return a vector `I` of the `true` indices or keys of `A`. If there are no such elements of `A`, return an empty array. To search for other kinds of values, pass a predicate as the first argument.

Indices or keys are of the same type as those returned by `keys(A)` and `pairs(A)`.

Examples

```
julia> A = [true, false, false, true]
4-element Array{Bool,1}:
 1
 0
 0
 1

julia> findall(A)
2-element Array{Int64,1}:
 1
 4

julia> A = [true false; false true]
2×2 Array{Bool,2}:
 1  0
 0  1

julia> findall(A)
2-element Array{CartesianIndex{2},1}:
 CartesianIndex(1, 1)
 CartesianIndex(2, 2)

julia> findall(falses(3))
Int64[]
```

`Base.findall` — Method

```
findall(f::Function, A)
```

Return a vector `I` of the indices or keys of `A` where `f(A[I])` returns `true`. If there are no such elements of `A`, return an empty array.

Indices or keys are of the same type as those returned by `keys(A)` and `pairs(A)`.

Examples

```
julia> x = [1, 3, 4]
3-element Array{Int64,1}:
 1
 3
 4

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

julia> A = [1 2 0; 3 4 0]
2×3 Array{Int64,2}:
 1  2  0
 3  4  0
julia> findall(isodd, A)
2-element Array{CartesianIndex{2},1}:
 CartesianIndex(1, 1)
 CartesianIndex(2, 1)

julia> findall(!iszero, A)
4-element Array{CartesianIndex{2},1}:
 CartesianIndex(1, 1)
 CartesianIndex(2, 1)
 CartesianIndex(1, 2)
 CartesianIndex(2, 2)

julia> d = Dict(:A => 10, :B => -1, :C => 0)
Dict{Symbol,Int64} with 3 entries:
  :A => 10
  :B => -1
  :C => 0

julia> findall(x -> x >= 0, d)
2-element Array{Symbol,1}:
 :A
 :C
```

Base.findfirst — Method

```
findfirst(A)
```

Return the index or key of the first `true` value in `A`. Return `nothing` if no such value is found. To search for other kinds of values, pass a predicate as the first argument.

Indices or keys are of the same type as those returned by `keys(A)` and `pairs(A)`.

Examples

```
julia> A = [false, false, true, false]
4-element Array{Bool,1}:
 0
 0
 1
 0

julia> findfirst(A)
3

julia> findfirst(falses(3)) # returns nothing, but not printed in the REPL

julia> A = [false false; true false]
2×2 Array{Bool,2}:
 0  0
 1  0

julia> findfirst(A)
CartesianIndex(2, 1)
```

Base.findfirst — Method

```
findfirst(predicate::Function, A)
```

Return the index or key of the first element of `A` for which `predicate` returns `true`. Return `nothing` if there is no such element.

Indices or keys are of the same type as those returned by `keys(A)` and `pairs(A)`.

Examples

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

julia> findfirst(iseven, A)
2

julia> findfirst(x -> x>10, A) # returns nothing, but not printed in the REPL

julia> findfirst(isequal(4), A)
2

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

julia> findfirst(iseven, A)
CartesianIndex(2, 1)
```

`Base.findlast` — Method

```
findlast(A)
```

Return the index or key of the last `true` value in `A`. Return `nothing` if there is no `true` value in `A`.

Indices or keys are of the same type as those returned by `keys(A)` and `pairs(A)`.

Examples

```
julia> A = [true, false, true, false]
4-element Array{Bool,1}:
 1
 0
 1
 0
```

```
julia> findlast(A)
3

julia> A = falses(2,2);

julia> findlast(A) # returns nothing, but not printed in the REPL

julia> A = [true false; true false]
2×2 Array{Bool,2}:
 1  0
 1  0

julia> findlast(A)
CartesianIndex(2, 1)
```

`Base.findlast` — Method

```
findlast(predicate::Function, A)
```

Return the index or key of the last element of `A` for which `predicate` returns `true`. Return `nothing` if there is no such element.

Indices or keys are of the same type as those returned by `keys(A)` and `pairs(A)`.

Examples

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

julia> findlast(isodd, A)
3

julia> findlast(x -> x > 5, A) # returns nothing, but not printed in the REPL

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

```
 3  4

julia> findlast(isodd, A)
CartesianIndex(2, 1)
```

Base.findnext — Method

```
findnext(A, i)
```

Find the next index after or including `i` of a `true` element of `A`, or `nothing` if not found.

Indices are of the same type as those returned by `keys(A)` and `pairs(A)`.

Examples

```
julia> A = [false, false, true, false]
4-element Array{Bool,1}:
 0
 0
 1
 0

julia> findnext(A, 1)
3

julia> findnext(A, 4) # returns nothing, but not printed in the REPL

julia> A = [false false; true false]
2×2 Array{Bool,2}:
 0  0
 1  0

julia> findnext(A, CartesianIndex(1, 1))
CartesianIndex(2, 1)
```

Base.findnext — Method

```
findnext(predicate::Function, A, i)
```

Find the next index after or including `i` of an element of `A` for which `predicate` returns `true`, or `nothing` if not found.

Indices are of the same type as those returned by `keys(A)` and `pairs(A)`.

Examples

```
julia> A = [1, 4, 2, 2];

julia> findnext(isodd, A, 1)
1

julia> findnext(isodd, A, 2) # returns nothing, but not printed in the REPL

julia> A = [1 4; 2 2];

julia> findnext(isodd, A, CartesianIndex(1, 1))
CartesianIndex(1, 1)
```

`Base.findprev` — Method

```
findprev(A, i)
```

Find the previous index before or including `i` of a `true` element of `A`, or `nothing` if not found.

Indices are of the same type as those returned by `keys(A)` and `pairs(A)`.

Examples

```
julia> A = [false, false, true, true]
4-element Array{Bool,1}:
 0
 0
 1
 1

julia> findprev(A, 3)
3

julia> findprev(A, 1) # returns nothing, but not printed in the REPL
```

```
julia> A = [false false; true true]
2×2 Array{Bool,2}:
 0  0
 1  1

julia> findprev(A, CartesianIndex(2, 1))
CartesianIndex(2, 1)
```

`Base.findprev` — Method

```
findprev(predicate::Function, A, i)
```

Find the previous index before or including `i` of an element of `A` for which `predicate` returns `true`, or `nothing` if not found.

Indices are of the same type as those returned by `keys(A)` and `pairs(A)`.

Examples

```
julia> A = [4, 6, 1, 2]
4-element Array{Int64,1}:
 4
 6
 1
 2

julia> findprev(isodd, A, 1) # returns nothing, but not printed in the REPL

julia> findprev(isodd, A, 3)
3

julia> A = [4 6; 1 2]
2×2 Array{Int64,2}:
 4  6
 1  2

julia> findprev(isodd, A, CartesianIndex(1, 2))
CartesianIndex(2, 1)
```

`Base.permutedims` — Function

```
permutedims(A::AbstractArray, perm)
```

Permute the dimensions of array `A`. `perm` is a vector specifying a permutation of length `ndims(A)`.

See also: `PermutedDimsArray`.

Examples

```
julia> A = reshape(Vector(1:8), (2,2,2))
2×2×2 Array{Int64,3}:
[:, :, 1] =
 1  3
 2  4

[:, :, 2] =
 5  7
 6  8

julia> permutedims(A, [3, 2, 1])
2×2×2 Array{Int64,3}:
[:, :, 1] =
 1  3
 5  7

[:, :, 2] =
 2  4
 6  8
```

```
permutedims(m::AbstractMatrix)
```

Permute the dimensions of the matrix `m`, by flipping the elements across the diagonal of the matrix. Differs from `LinearAlgebra`'s `transpose` in that the operation is not recursive.

Examples

```
julia> a = [1 2; 3 4];

julia> b = [5 6; 7 8];
```

```
julia> c = [9 10; 11 12];

julia> d = [13 14; 15 16];

julia> X = [[a] [b]; [c] [d]]
2×2 Array{Array{Int64,2},2}:
 [1 2; 3 4]     [5 6; 7 8]
 [9 10; 11 12]  [13 14; 15 16]

julia> permutedims(X)
2×2 Array{Array{Int64,2},2}:
 [1 2; 3 4]  [9 10; 11 12]
 [5 6; 7 8]  [13 14; 15 16]

julia> transpose(X)
2×2 Transpose{Transpose{Int64,Array{Int64,2}},Array{Array{Int64,2},2}}:
 [1 3; 2 4]  [9 11; 10 12]
 [5 7; 6 8]  [13 15; 14 16]
```

```
permutedims(v::AbstractVector)
```

Reshape vector `v` into a `1 × length(v)` row matrix. Differs from `LinearAlgebra`'s `transpose` in that the operation is not recursive.

Examples

```
julia> permutedims([1, 2, 3, 4])
1×4 Array{Int64,2}:
 1  2  3  4

julia> V = [[[1 2; 3 4]]; [[5 6; 7 8]]]
2-element Array{Array{Int64,2},1}:
 [1 2; 3 4]
 [5 6; 7 8]

julia> permutedims(V)
1×2 Array{Array{Int64,2},2}:
 [1 2; 3 4]  [5 6; 7 8]

julia> transpose(V)
1×2 Transpose{Transpose{Int64,Array{Int64,2}},Array{Array{Int64,2},1}}:
 [1 3; 2 4]  [5 7; 6 8]
```

`Base.permutedims!` — Function

```
permutedims!(dest, src, perm)
```

Permute the dimensions of array `src` and store the result in the array `dest`. `perm` is a vector specifying a permutation of length `ndims(src)`. The preallocated array `dest` should have `size(dest) == size(src)[perm]` and is completely overwritten. No in-place permutation is supported and unexpected results will happen if `src` and `dest` have overlapping memory regions.

See also `permutedims`.

`Base.PermutedDimsArrays.PermutedDimsArray` — Type

```
PermutedDimsArray(A, perm) -> B
```

Given an AbstractArray `A`, create a view `B` such that the dimensions appear to be permuted. Similar to `permutedims`, except that no copying occurs (`B` shares storage with `A`).

See also: `permutedims`.

Examples

```
julia> A = rand(3,5,4);

julia> B = PermutedDimsArray(A, (3,1,2));

julia> size(B)
(4, 3, 5)

julia> B[3,1,2] == A[1,2,3]
true
```

`Base.promote_shape` — Function

```
promote_shape(s1, s2)
```

Check two array shapes for compatibility, allowing trailing singleton dimensions, and return whichever shape has more dimensions.

Examples

```
julia> a = fill(1, (3,4,1,1,1));

julia> b = fill(1, (3,4));

julia> promote_shape(a,b)
(Base.OneTo(3), Base.OneTo(4), Base.OneTo(1), Base.OneTo(1), Base.OneTo(1))

julia> promote_shape((2,3,1,4), (2, 3, 1, 4, 1))
(2, 3, 1, 4, 1)
```

Array functions

`Base.accumulate` — Function

```
accumulate(op, A; dims::Integer, [init])
```

Cumulative operation `op` along the dimension `dims` of `A` (providing `dims` is optional for vectors). An initial value `init` may optionally be provided by a keyword argument. See also `accumulate!` to use a preallocated output array, both for performance and to control the precision of the output (e.g. to avoid overflow). For common operations there are specialized variants of `accumulate`, see: `cumsum`, `cumprod`

Julia 1.5

`accumulate` on a non-array iterator requires at least Julia 1.5.

Examples

```
julia> accumulate(+, [1,2,3])
3-element Array{Int64,1}:
 1
 3
 6

julia> accumulate(*, [1,2,3])
3-element Array{Int64,1}:
 1
 2
 6

julia> accumulate(+, [1,2,3]; init=100)
3-element Array{Int64,1}:
 101
 103
 106

julia> accumulate(min, [1,2,-1]; init=0)
3-element Array{Int64,1}:
  0
  0
```

```
 -1

julia> accumulate(+, fill(1, 3, 3), dims=1)
3×3 Array{Int64,2}:
 1  1  1
 2  2  2
 3  3  3

julia> accumulate(+, fill(1, 3, 3), dims=2)
3×3 Array{Int64,2}:
 1  2  3
 1  2  3
 1  2  3
```

`Base.accumulate!` — Function

```
accumulate!(op, B, A; [dims], [init])
```

Cumulative operation `op` on `A` along the dimension `dims`, storing the result in `B`. Providing `dims` is optional for vectors. If the keyword argument `init` is given, its value is used to instantiate the accumulation. See also `accumulate`.

Examples

```
julia> x = [1, 0, 2, 0, 3];

julia> y = [0, 0, 0, 0, 0];

julia> accumulate!(+, y, x);

julia> y
5-element Array{Int64,1}:
 1
 1
 3
 3
 6

julia> A = [1 2; 3 4];

julia> B = [0 0; 0 0];
```

```
julia> accumulate!(-, B, A, dims=1);

julia> B
2×2 Array{Int64,2}:
  1   2
 -2  -2

julia> accumulate!(-, B, A, dims=2);

julia> B
2×2 Array{Int64,2}:
 1  -1
 3  -1
```

`Base.cumprod` — Function

```
cumprod(A; dims::Integer)
```

Cumulative product along the dimension `dim`. See also `cumprod!` to use a preallocated output array, both for performance and to control the precision of the output (e.g. to avoid overflow).

Examples

```
julia> a = [1 2 3; 4 5 6]
2×3 Array{Int64,2}:
 1  2  3
 4  5  6

julia> cumprod(a, dims=1)
2×3 Array{Int64,2}:
 1   2   3
 4  10  18

julia> cumprod(a, dims=2)
2×3 Array{Int64,2}:
 1   2    6
 4  20  120
```

```
cumprod(itr)
```

Cumulative product of an iterator. See also `cumprod!` to use a preallocated output array, both for performance and to control the precision of the output (e.g. to avoid overflow).

> **Julia 1.5**
>
> `cumprod` on a non-array iterator requires at least Julia 1.5.

Examples

```
julia> cumprod(fill(1//2, 3))
3-element Array{Rational{Int64},1}:
 1//2
 1//4
 1//8

julia> cumprod([fill(1//3, 2, 2) for i in 1:3])
3-element Array{Array{Rational{Int64},2},1}:
 [1//3 1//3; 1//3 1//3]
 [2//9 2//9; 2//9 2//9]
 [4//27 4//27; 4//27 4//27]

julia> cumprod((1, 2, 1))
(1, 2, 2)

julia> cumprod(x^2 for x in 1:3)
3-element Array{Int64,1}:
  1
  4
 36
```

`Base.cumprod!` — Function

```
cumprod!(B, A; dims::Integer)
```

Cumulative product of `A` along the dimension `dims`, storing the result in `B`. See also `cumprod`.

```
cumprod!(y::AbstractVector, x::AbstractVector)
```

Cumulative product of a vector `x`, storing the result in `y`. See also `cumprod`.

`Base.cumsum` — Function

```
cumsum(A; dims::Integer)
```

Cumulative sum along the dimension `dims`. See also `cumsum!` to use a preallocated output array, both for performance and to control the precision of the output (e.g. to avoid overflow).

Examples

```
julia> a = [1 2 3; 4 5 6]
2×3 Array{Int64,2}:
 1  2  3
 4  5  6

julia> cumsum(a, dims=1)
2×3 Array{Int64,2}:
 1  2  3
 5  7  9

julia> cumsum(a, dims=2)
2×3 Array{Int64,2}:
 1  3   6
 4  9  15
```

> **Note**
>
> The return array's `eltype` is `Int` for signed integers of less than system word size and `UInt` for unsigned integers of less than system word size. To preserve `eltype` of arrays with small signed or unsigned integer `accumulate(+, A)` should be used.
>
> ```
> julia> cumsum(Int8[100, 28])
> 2-element Array{Int64,1}:
>  100
>  128
>
> julia> accumulate(+,Int8[100, 28])
> 2-element Array{Int8,1}:
>   100
> ```

```
-128
```

In the former case, the integers are widened to system word size and therefore the result is `Int64[100, 128]`. In the latter case, no such widening happens and integer overflow results in `Int8[100, -128]`.

```
cumsum(itr)
```

Cumulative sum an iterator. See also `cumsum!` to use a preallocated output array, both for performance and to control the precision of the output (e.g. to avoid overflow).

Julia 1.5

`cumsum` on a non-array iterator requires at least Julia 1.5.

Examples

```
julia> cumsum([1, 1, 1])
3-element Array{Int64,1}:
 1
 2
 3

julia> cumsum([fill(1, 2) for i in 1:3])
3-element Array{Array{Int64,1},1}:
 [1, 1]
 [2, 2]
 [3, 3]

julia> cumsum((1, 1, 1))
(1, 2, 3)

julia> cumsum(x^2 for x in 1:3)
3-element Array{Int64,1}:
  1
  5
 14
```

`Base.cumsum!` — Function

```
cumsum!(B, A; dims::Integer)
```

Cumulative sum of `A` along the dimension `dims`, storing the result in `B`. See also `cumsum`.

`Base.diff` — Function

```
diff(A::AbstractVector)
diff(A::AbstractArray; dims::Integer)
```

Finite difference operator on a vector or a multidimensional array `A`. In the latter case the dimension to operate on needs to be specified with the `dims` keyword argument.

> Julia 1.1
>
> `diff` for arrays with dimension higher than 2 requires at least Julia 1.1.

Examples

```
julia> a = [2 4; 6 16]
2×2 Array{Int64,2}:
 2   4
 6  16

julia> diff(a, dims=2)
2×1 Array{Int64,2}:
  2
 10

julia> diff(vec(a))
3-element Array{Int64,1}:
  4
 -2
 12
```

`Base.repeat` — Function

```
repeat(A::AbstractArray, counts::Integer...)
```

Construct an array by repeating array `A` a given number of times in each dimension, specified by `counts`.

Examples

```
julia> repeat([1, 2, 3], 2)
6-element Array{Int64,1}:
 1
 2
 3
 1
 2
 3

julia> repeat([1, 2, 3], 2, 3)
6×3 Array{Int64,2}:
 1  1  1
 2  2  2
 3  3  3
 1  1  1
 2  2  2
 3  3  3
```

```
repeat(A::AbstractArray; inner=ntuple(x->1, ndims(A)), outer=ntuple(x->1, ndims(
```

Construct an array by repeating the entries of `A`. The i-th element of `inner` specifies the number of times that the individual entries of the i-th dimension of `A` should be repeated. The i-th element of `outer` specifies the number of times that a slice along the i-th dimension of `A` should be repeated. If `inner` or `outer` are omitted, no repetition is performed.

Examples

```
julia> repeat(1:2, inner=2)
4-element Array{Int64,1}:
 1
 1
```

```
 2
 2

julia> repeat(1:2, outer=2)
4-element Array{Int64,1}:
 1
 2
 1
 2

julia> repeat([1 2; 3 4], inner=(2, 1), outer=(1, 3))
4×6 Array{Int64,2}:
 1  2  1  2  1  2
 1  2  1  2  1  2
 3  4  3  4  3  4
 3  4  3  4  3  4
```

```
repeat(s::AbstractString, r::Integer)
```

Repeat a string `r` times. This can be written as `s^r`.

See also: `^`

Examples

```
julia> repeat("ha", 3)
"hahaha"
```

```
repeat(c::AbstractChar, r::Integer) -> String
```

Repeat a character `r` times. This can equivalently be accomplished by calling `c^r`.

Examples

```
julia> repeat('A', 3)
"AAA"
```

`Base.rot180` — Function

```
rot180(A)
```

Rotate matrix `A` 180 degrees.

Examples

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

julia> rot180(a)
2×2 Array{Int64,2}:
 4  3
 2  1
```

```
rot180(A, k)
```

Rotate matrix `A` 180 degrees an integer `k` number of times. If `k` is even, this is equivalent to a `copy`.

Examples

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

julia> rot180(a,1)
2×2 Array{Int64,2}:
 4  3
 2  1

julia> rot180(a,2)
2×2 Array{Int64,2}:
 1  2
 3  4
```

`Base.rotl90` — Function

```
rotl90(A)
```

Rotate matrix `A` left 90 degrees.

Examples

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

julia> rotl90(a)
2×2 Array{Int64,2}:
 2  4
 1  3
```

```
rotl90(A, k)
```

Left-rotate matrix `A` 90 degrees counterclockwise an integer `k` number of times. If `k` is a multiple of four (including zero), this is equivalent to a `copy`.

Examples

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

julia> rotl90(a,1)
2×2 Array{Int64,2}:
 2  4
 1  3

julia> rotl90(a,2)
2×2 Array{Int64,2}:
 4  3
 2  1
```

```
julia> rotl90(a,3)
2×2 Array{Int64,2}:
 3  1
 4  2

julia> rotl90(a,4)
2×2 Array{Int64,2}:
 1  2
 3  4
```

`Base.rotr90` — Function

```
rotr90(A)
```

Rotate matrix `A` right 90 degrees.

Examples

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

julia> rotr90(a)
2×2 Array{Int64,2}:
 3  1
 4  2
```

```
rotr90(A, k)
```

Right-rotate matrix `A` 90 degrees clockwise an integer `k` number of times. If `k` is a multiple of four (including zero), this is equivalent to a `copy`.

Examples

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

```
julia> rotr90(a,1)
2×2 Array{Int64,2}:
 3  1
 4  2

julia> rotr90(a,2)
2×2 Array{Int64,2}:
 4  3
 2  1

julia> rotr90(a,3)
2×2 Array{Int64,2}:
 2  4
 1  3

julia> rotr90(a,4)
2×2 Array{Int64,2}:
 1  2
 3  4
```

`Base.mapslices` — Function

```
mapslices(f, A; dims)
```

Transform the given dimensions of array `A` using function `f`. `f` is called on each slice of `A` of the form `A[...,:,...,:,...]`. `dims` is an integer vector specifying where the colons go in this expression. The results are concatenated along the remaining dimensions. For example, if `dims` is `[1,2]` and `A` is 4-dimensional, `f` is called on `A[:,:,i,j]` for all `i` and `j`.

Examples

```
julia> a = reshape(Vector(1:16),(2,2,2,2))
2×2×2×2 Array{Int64,4}:
[:, :, 1, 1] =
 1  3
 2  4

[:, :, 2, 1] =
 5  7
 6  8
```

```
[:, :, 1, 2] =
  9  11
 10  12

[:, :, 2, 2] =
 13  15
 14  16

julia> mapslices(sum, a, dims = [1,2])
1×1×2×2 Array{Int64,4}:
[:, :, 1, 1] =
 10

[:, :, 2, 1] =
 26

[:, :, 1, 2] =
 42

[:, :, 2, 2] =
 58
```

`Base.eachrow` — Function

```
eachrow(A::AbstractVecOrMat)
```

Create a generator that iterates over the first dimension of vector or matrix `A`, returning the rows as `AbstractVector` views.

See also `eachcol` and `eachslice`.

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

Example

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

```
 1  2
 3  4

julia> first(eachrow(a))
2-element view(::Array{Int64,2}, 1, :) with eltype Int64:
 1
 2

julia> collect(eachrow(a))
2-element Array{SubArray{Int64,1,Array{Int64,2},Tuple{Int64,Base.Slice{Base.One
 [1, 2]
 [3, 4]
```

`Base.eachcol` — Function

```
eachcol(A::AbstractVecOrMat)
```

Create a generator that iterates over the second dimension of matrix `A`, returning the columns as `AbstractVector` views.

See also `eachrow` and `eachslice`.

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

Example

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

julia> first(eachcol(a))
2-element view(::Array{Int64,2}, :, 1) with eltype Int64:
 1
 3

julia> collect(eachcol(a))
2-element Array{SubArray{Int64,1,Array{Int64,2},Tuple{Base.Slice{Base.OneTo{Int
```

```
 [1, 3]
 [2, 4]
```

`Base.eachslice` — Function

```
eachslice(A::AbstractArray; dims)
```

Create a generator that iterates over dimensions `dims` of `A`, returning views that select all the data from the other dimensions in `A`.

Only a single dimension in `dims` is currently supported. Equivalent to `(view(A,:,:,...,i,:,:...)) for i in axes(A, dims))`, where `i` is in position `dims`.

See also `eachrow`, `eachcol`, and `selectdim`.

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

Combinatorics

`Base.invperm` — Function

```
invperm(v)
```

Return the inverse permutation of `v`. If `B = A[v]`, then `A == B[invperm(v)]`.

Examples

```
julia> v = [2; 4; 3; 1];

julia> invperm(v)
4-element Array{Int64,1}:
 4
 1
 3
```

```
 2

julia> A = ['a','b','c','d'];

julia> B = A[v]
4-element Array{Char,1}:
 'b': ASCII/Unicode U+0062 (category Ll: Letter, lowercase)
 'd': ASCII/Unicode U+0064 (category Ll: Letter, lowercase)
 'c': ASCII/Unicode U+0063 (category Ll: Letter, lowercase)
 'a': ASCII/Unicode U+0061 (category Ll: Letter, lowercase)

julia> B[invperm(v)]
4-element Array{Char,1}:
 'a': ASCII/Unicode U+0061 (category Ll: Letter, lowercase)
 'b': ASCII/Unicode U+0062 (category Ll: Letter, lowercase)
 'c': ASCII/Unicode U+0063 (category Ll: Letter, lowercase)
 'd': ASCII/Unicode U+0064 (category Ll: Letter, lowercase)
```

`Base.isperm` — Function

```
isperm(v) -> Bool
```

Return `true` if `v` is a valid permutation.

Examples

```
julia> isperm([1; 2])
true

julia> isperm([1; 3])
false
```

`Base.permute!` — Method

```
permute!(v, p)
```

Permute vector `v` in-place, according to permutation `p`. No checking is done to verify that `p` is a permutation.

To return a new permutation, use `v[p]`. Note that this is generally faster than `permute!(v,p)` for large vectors.

See also `invpermute!`.

Examples

```
julia> A = [1, 1, 3, 4];

julia> perm = [2, 4, 3, 1];

julia> permute!(A, perm);

julia> A
4-element Array{Int64,1}:
 1
 4
 3
 1
```

`Base.invpermute!` — Function

```
invpermute!(v, p)
```

Like `permute!`, but the inverse of the given permutation is applied.

Examples

```
julia> A = [1, 1, 3, 4];

julia> perm = [2, 4, 3, 1];

julia> invpermute!(A, perm);

julia> A
4-element Array{Int64,1}:
 4
 1
 3
 1
```

`Base.reverse` — Method

```
reverse(v [, start=1 [, stop=length(v) ]] )
```

Return a copy of `v` reversed from start to stop. See also `Iterators.reverse` for reverse-order iteration without making a copy.

Examples

```
julia> A = Vector(1:5)
5-element Array{Int64,1}:
 1
 2
 3
 4
 5

julia> reverse(A)
5-element Array{Int64,1}:
 5
 4
 3
 2
 1

julia> reverse(A, 1, 4)
5-element Array{Int64,1}:
 4
 3
 2
 1
 5

julia> reverse(A, 3, 5)
5-element Array{Int64,1}:
 1
 2
 5
 4
 3
```

```
reverse(A; dims::Integer)
```

Reverse `A` in dimension `dims`.

Examples

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

julia> reverse(b, dims=2)
2×2 Array{Int64,2}:
 2  1
 4  3
```

`Base.reverseind` — Function

```
reverseind(v, i)
```

Given an index `i` in `reverse(v)`, return the corresponding index in `v` so that `v[reverseind(v,i)] == reverse(v)[i]`. (This can be nontrivial in cases where `v` contains non-ASCII characters.)

Examples

```
julia> r = reverse("Julia")
"ailuJ"

julia> for i in 1:length(r)
           print(r[reverseind("Julia", i)])
       end
Julia
```

`Base.reverse!` — Function

```
reverse!(v [, start=1 [, stop=length(v) ]]) -> v
```

In-place version of `reverse`.

Examples

```
julia> A = Vector(1:5)
5-element Array{Int64,1}:
 1
 2
 3
 4
 5

julia> reverse!(A);

julia> A
5-element Array{Int64,1}:
 5
 4
 3
 2
 1
```

« Strings Tasks »