Collections and Data Structures

Iteration

Sequential iteration is implemented by the `iterate` function. The general `for` loop:

```
for i in iter   # or  "for i = iter"
    # body
end
```

is translated into:

```
next = iterate(iter)
while next !== nothing
    (i, state) = next
    # body
    next = iterate(iter, state)
end
```

The `state` object may be anything, and should be chosen appropriately for each iterable type. See the manual section on the iteration interface for more details about defining a custom iterable type.

`Base.iterate` — Function

```
iterate(iter [, state]) -> Union{Nothing, Tuple{Any, Any}}
```

Advance the iterator to obtain the next element. If no elements remain, `nothing` should be returned. Otherwise, a 2-tuple of the next element and the new iteration state should be returned.

`Base.IteratorSize` — Type

```
IteratorSize(itertype::Type) -> IteratorSize
```

Given the type of an iterator, return one of the following values:

- `SizeUnknown()` if the length (number of elements) cannot be determined in advance.
- `HasLength()` if there is a fixed, finite length.
- `HasShape{N}()` if there is a known length plus a notion of multidimensional shape (as for an array). In this case `N` should give the number of dimensions, and the `axes` function is valid for the iterator.
- `IsInfinite()` if the iterator yields values forever.

The default value (for iterators that do not define this function) is `HasLength()`. This means that most iterators are assumed to implement `length`.

This trait is generally used to select between algorithms that pre-allocate space for their result, and algorithms that resize their result incrementally.

```
julia> Base.IteratorSize(1:5)
Base.HasShape{1}()

julia> Base.IteratorSize((2,3))
Base.HasLength()
```

`Base.IteratorEltype` — Type

```
IteratorEltype(itertype::Type) -> IteratorEltype
```

Given the type of an iterator, return one of the following values:

- `EltypeUnknown()` if the type of elements yielded by the iterator is not known in advance.
- `HasEltype()` if the element type is known, and `eltype` would return a meaningful value.

`HasEltype()` is the default, since iterators are assumed to implement `eltype`.

This trait is generally used to select between algorithms that pre-allocate a specific type of result, and algorithms that pick a result type based on the types of yielded values.

```
julia> Base.IteratorEltype(1:5)
Base.HasEltype()
```

Fully implemented by:

- AbstractRange
- UnitRange
- Tuple
- Number
- AbstractArray
- BitSet
- IdDict
- Dict
- WeakKeyDict
- EachLine
- AbstractString
- Set
- Pair
- NamedTuple

Constructors and Types

Base.AbstractRange — Type

```
AbstractRange{T}
```

Supertype for ranges with elements of type `T`. `UnitRange` and other types are subtypes of this.

Base.OrdinalRange — Type

```
OrdinalRange{T, S} <: AbstractRange{T}
```

Supertype for ordinal ranges with elements of type `T` with spacing(s) of type `S`. The steps should be always-exact multiples of `oneunit`, and `T` should be a "discrete" type, which cannot have values smaller than `oneunit`. For example, `Integer` or `Date` types would qualify, whereas `Float64` would not (since this type can represent values smaller than `oneunit(Float64)`. `UnitRange`, `StepRange`, and other types are subtypes of this.

Base.AbstractUnitRange — Type

```
AbstractUnitRange{T} <: OrdinalRange{T, T}
```

Supertype for ranges with a step size of `oneunit(T)` with elements of type `T`. `UnitRange` and other types are subtypes of this.

Base.StepRange — Type

```
StepRange{T, S} <: OrdinalRange{T, S}
```

Ranges with elements of type `T` with spacing of type `S`. The step between each element is constant, and the range is defined in terms of a `start` and `stop` of type `T` and a `step` of type `S`. Neither `T` nor `S` should be floating point types. The syntax `a:b:c` with `b > 1` and `a`, `b`, and `c` all integers creates a `StepRange`.

Examples

```
julia> collect(StepRange(1, Int8(2), 10))
5-element Array{Int64,1}:
 1
 3
 5
 7
 9

julia> typeof(StepRange(1, Int8(2), 10))
StepRange{Int64,Int8}

julia> typeof(1:3:6)
StepRange{Int64,Int64}
```

Base.UnitRange — Type

```
UnitRange{T<:Real}
```

A range parameterized by a `start` and `stop` of type `T`, filled with elements spaced by `1` from

`start` until `stop` is exceeded. The syntax `a:b` with `a` and `b` both `Integers` creates a `UnitRange`.

Examples

```
julia> collect(UnitRange(2.3, 5.2))
3-element Array{Float64,1}:
 2.3
 3.3
 4.3

julia> typeof(1:10)
UnitRange{Int64}
```

`Base.LinRange` — Type

```
LinRange{T}
```

A range with `len` linearly spaced elements between its `start` and `stop`. The size of the spacing is controlled by `len`, which must be an `Int`.

Examples

```
julia> LinRange(1.5, 5.5, 9)
9-element LinRange{Float64}:
 1.5,2.0,2.5,3.0,3.5,4.0,4.5,5.0,5.5
```

Compared to using `range`, directly constructing a `LinRange` should have less overhead but won't try to correct for floating point errors:

```
julia> collect(range(-0.1, 0.3, length=5))
5-element Array{Float64,1}:
 -0.1
  0.0
  0.1
  0.2
  0.3

julia> collect(LinRange(-0.1, 0.3, 5))
5-element Array{Float64,1}:
 -0.1
```

```
-1.3877787807814457e-17
 0.09999999999999999
 0.19999999999999998
 0.3
```

General Collections

`Base.isempty` — Function

```
isempty(collection) -> Bool
```

Determine whether a collection is empty (has no elements).

Examples

```
julia> isempty([])
true

julia> isempty([1 2 3])
false
```

```
isempty(condition)
```

Return `true` if no tasks are waiting on the condition, `false` otherwise.

`Base.empty!` — Function

```
empty!(collection) -> collection
```

Remove all elements from a `collection`.

Examples

```
julia> A = Dict("a" => 1, "b" => 2)
Dict{String,Int64} with 2 entries:
  "b" => 2
```

```
  "a" => 1

julia> empty!(A);

julia> A
Dict{String,Int64}()
```

`Base.length` — Function

```
length(collection) -> Integer
```

Return the number of elements in the collection.

Use `lastindex` to get the last valid index of an indexable collection.

Examples

```
julia> length(1:5)
5

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

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

Fully implemented by:

- `AbstractRange`
- `UnitRange`
- `Tuple`
- `Number`
- `AbstractArray`
- `BitSet`
- `IdDict`
- `Dict`
- `WeakKeyDict`
- `AbstractString`

- `Set`
- `NamedTuple`

Iterable Collections

`Base.in` — Function

```
in(item, collection) -> Bool
∈(item, collection) -> Bool
∋(collection, item) -> Bool
```

Determine whether an item is in the given collection, in the sense that it is `==` to one of the values generated by iterating over the collection. Returns a `Bool` value, except if `item` is `missing` or `collection` contains `missing` but not `item`, in which case `missing` is returned (three-valued logic, matching the behavior of `any` and `==`).

Some collections follow a slightly different definition. For example, `Set`s check whether the item `isequal` to one of the elements. `Dict`s look for `key=>value` pairs, and the key is compared using `isequal`. To test for the presence of a key in a dictionary, use `haskey` or `k in keys(dict)`. For these collections, the result is always a `Bool` and never `missing`.

To determine whether an item is not in a given collection, see `:∉`. You may also negate the `in` by doing `!(a in b)` which is logically similar to "not in".

When broadcasting with `in.(items, collection)` or `items .∈ collection`, both `item` and `collection` are broadcasted over, which is often not what is intended. For example, if both arguments are vectors (and the dimensions match), the result is a vector indicating whether each value in collection `items` is `in` the value at the corresponding position in `collection`. To get a vector indicating whether each value in `items` is in `collection`, wrap `collection` in a tuple or a `Ref` like this: `in.(items, Ref(collection))` or `items .∈ Ref(collection)`.

Examples

```
julia> a = 1:3:20
1:3:19

julia> 4 in a
true

julia> 5 in a
```

```
false

julia> missing in [1, 2]
missing

julia> 1 in [2, missing]
missing

julia> 1 in [1, missing]
true

julia> missing in Set([1, 2])
false

julia> !(21 in a)
true

julia> !(19 in a)
false

julia> [1, 2] .∈ [2, 3]
2-element BitArray{1}:
 0
 0

julia> [1, 2] .∈ ([2, 3],)
2-element BitArray{1}:
 0
 1
```

`Base.:∉` — Function

```
∉(item, collection) -> Bool
∌(collection, item) -> Bool
```

Negation of `∈` and `∋`, i.e. checks that `item` is not in `collection`.

When broadcasting with `items .∉ collection`, both `item` and `collection` are broadcasted over, which is often not what is intended. For example, if both arguments are vectors (and the dimensions match), the result is a vector indicating whether each value in collection `items` is not in the value at the corresponding position in `collection`. To get a vector indicating whether each value in `items` is not in `collection`, wrap `collection` in a tuple or a `Ref` like this: `items .∉`

`Ref(collection)`.

Examples

```
julia> 1 ∉ 2:4
true

julia> 1 ∉ 1:3
false

julia> [1, 2] .∉ [2, 3]
2-element BitArray{1}:
 1
 1

julia> [1, 2] .∉ ([2, 3],)
2-element BitArray{1}:
 1
 0
```

`Base.eltype` — Function

```
eltype(type)
```

Determine the type of the elements generated by iterating a collection of the given `type`. For dictionary types, this will be a `Pair{KeyType,ValType}`. The definition `eltype(x) = eltype(typeof(x))` is provided for convenience so that instances can be passed instead of types. However the form that accepts a type argument should be defined for new types.

Examples

```
julia> eltype(fill(1f0, (2,2)))
Float32

julia> eltype(fill(0x1, (2,2)))
UInt8
```

`Base.indexin` — Function

```
indexin(a, b)
```

Return an array containing the first index in `b` for each value in `a` that is a member of `b`. The output array contains `nothing` wherever `a` is not a member of `b`.

Examples

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

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

julia> indexin(a, b)
6-element Array{Union{Nothing, Int64},1}:
 1
 2
 3
 2
  nothing
 1

julia> indexin(b, a)
3-element Array{Union{Nothing, Int64},1}:
 1
 2
 3
```

`Base.unique` — Function

```
unique(itr)
```

Return an array containing only the unique elements of collection `itr`, as determined by `isequal`, in the order that the first of each set of equivalent elements originally appears. The element type of the input is preserved.

Examples

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

```
 2
 6

julia> unique(Real[1, 1.0, 2])
2-element Array{Real,1}:
 1
 2
```

```
unique(f, itr)
```

Returns an array containing one value from `itr` for each unique value produced by `f` applied to elements of `itr`.

Examples

```
julia> unique(x -> x^2, [1, -1, 3, -3, 4])
3-element Array{Int64,1}:
 1
 3
 4
```

```
unique(A::AbstractArray; dims::Int)
```

Return unique regions of `A` along dimension `dims`.

Examples

```
julia> A = map(isodd, reshape(Vector(1:8), (2,2,2)))
2×2×2 Array{Bool,3}:
[:, :, 1] =
 1  1
 0  0

[:, :, 2] =
 1  1
 0  0

julia> unique(A)
2-element Array{Bool,1}:
 1
```

```
 0

julia> unique(A, dims=2)
2×1×2 Array{Bool,3}:
[:, :, 1] =
 1
 0

[:, :, 2] =
 1
 0

julia> unique(A, dims=3)
2×2×1 Array{Bool,3}:
[:, :, 1] =
 1  1
 0  0
```

`Base.unique!` — Function

```
unique!(f, A::AbstractVector)
```

Selects one value from `A` for each unique value produced by `f` applied to elements of `A` , then return the modified A.

> **Julia 1.1**
>
> This method is available as of Julia 1.1.

Examples

```
julia> unique!(x -> x^2, [1, -1, 3, -3, 4])
3-element Array{Int64,1}:
 1
 3
 4

julia> unique!(n -> n%3, [5, 1, 8, 9, 3, 4, 10, 7, 2, 6])
3-element Array{Int64,1}:
 5
```

```
 1
 9

julia> unique!(iseven, [2, 3, 5, 7, 9])
2-element Array{Int64,1}:
 2
 3
```

```
unique!(A::AbstractVector)
```

Remove duplicate items as determined by `isequal`, then return the modified `A`. `unique!` will return the elements of `A` in the order that they occur. If you do not care about the order of the returned data, then calling `(sort!(A); unique!(A))` will be much more efficient as long as the elements of `A` can be sorted.

Examples

```
julia> unique!([1, 1, 1])
1-element Array{Int64,1}:
 1

julia> A = [7, 3, 2, 3, 7, 5];

julia> unique!(A)
4-element Array{Int64,1}:
 7
 3
 2
 5

julia> B = [7, 6, 42, 6, 7, 42];

julia> sort!(B);  # unique! is able to process sorted data much more efficientl

julia> unique!(B)
3-element Array{Int64,1}:
  6
  7
 42
```

`Base.allunique` — Function

```
allunique(itr) -> Bool
```

Return `true` if all values from `itr` are distinct when compared with `isequal`.

Examples

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

julia> allunique([a, a])
false
```

`Base.reduce` — Method

```
reduce(op, itr; [init])
```

Reduce the given collection `itr` with the given binary operator `op`. If provided, the initial value `init` must be a neutral element for `op` that will be returned for empty collections. It is unspecified whether `init` is used for non-empty collections.

For empty collections, providing `init` will be necessary, except for some special cases (e.g. when `op` is one of `+`, `*`, `max`, `min`, `&`, `|`) when Julia can determine the neutral element of `op`.

Reductions for certain commonly-used operators may have special implementations, and should be used instead: `maximum(itr)`, `minimum(itr)`, `sum(itr)`, `prod(itr)`, `any(itr)`, `all(itr)`.

The associativity of the reduction is implementation dependent. This means that you can't use non-associative operations like `-` because it is undefined whether `reduce(-,[1,2,3])` should be evaluated as `(1-2)-3` or `1-(2-3)`. Use `foldl` or `foldr` instead for guaranteed left or right associativity.

Some operations accumulate error. Parallelism will be easier if the reduction can be executed in groups. Future versions of Julia might change the algorithm. Note that the elements are not reordered if you use an ordered collection.

Examples

```
julia> reduce(*, [2; 3; 4])
24

julia> reduce(*, [2; 3; 4]; init=-1)
-24
```

Base.foldl — Method

```
foldl(op, itr; [init])
```

Like `reduce`, but with guaranteed left associativity. If provided, the keyword argument `init` will be used exactly once. In general, it will be necessary to provide `init` to work with empty collections.

Examples

```
julia> foldl(=>, 1:4)
((1 => 2) => 3) => 4

julia> foldl(=>, 1:4; init=0)
(((0 => 1) => 2) => 3) => 4
```

Base.foldr — Method

```
foldr(op, itr; [init])
```

Like `reduce`, but with guaranteed right associativity. If provided, the keyword argument `init` will be used exactly once. In general, it will be necessary to provide `init` to work with empty collections.

Examples

```
julia> foldr(=>, 1:4)
1 => (2 => (3 => 4))
```

```
julia> foldr(=>, 1:4; init=0)
1 => (2 => (3 => (4 => 0)))
```

`Base.maximum` — Function

```
maximum(f, itr)
```

Returns the largest result of calling function `f` on each element of `itr`.

Examples

```
julia> maximum(length, ["Julion", "Julia", "Jule"])
6
```

```
maximum(itr)
```

Returns the largest element in a collection.

Examples

```
julia> maximum(-20.5:10)
9.5

julia> maximum([1,2,3])
3
```

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

Compute the maximum value of an array over the given dimensions. See also the `max(a,b)` function to take the maximum of two or more arguments, which can be applied elementwise to arrays via `max.(a,b)`.

Examples

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

```
 3  4

julia> maximum(A, dims=1)
1×2 Array{Int64,2}:
 3  4

julia> maximum(A, dims=2)
2×1 Array{Int64,2}:
 2
 4
```

`Base.maximum!` — Function

```
maximum!(r, A)
```

Compute the maximum value of `A` over the singleton dimensions of `r`, and write results to `r`.

Examples

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

julia> maximum!([1; 1], A)
2-element Array{Int64,1}:
 2
 4

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

`Base.minimum` — Function

```
minimum(f, itr)
```

Returns the smallest result of calling function `f` on each element of `itr`.

Examples

```
julia> minimum(length, ["Julion", "Julia", "Jule"])
4
```

```
minimum(itr)
```

Returns the smallest element in a collection.

Examples

```
julia> minimum(-20.5:10)
-20.5

julia> minimum([1,2,3])
1
```

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

Compute the minimum value of an array over the given dimensions. See also the `min(a,b)` function to take the minimum of two or more arguments, which can be applied elementwise to arrays via `min.(a,b)`.

Examples

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

julia> minimum(A, dims=1)
1×2 Array{Int64,2}:
 1  2

julia> minimum(A, dims=2)
2×1 Array{Int64,2}:
 1
 3
```

Base.minimum! — Function

```
minimum!(r, A)
```

Compute the minimum value of `A` over the singleton dimensions of `r`, and write results to `r`.

Examples

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

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

julia> minimum!([1 1], A)
1×2 Array{Int64,2}:
 1  2
```

Base.extrema — Function

```
extrema(itr) -> Tuple
```

Compute both the minimum and maximum element in a single pass, and return them as a 2-tuple.

Examples

```
julia> extrema(2:10)
(2, 10)

julia> extrema([9,pi,4.5])
(3.141592653589793, 9.0)
```

```
extrema(f, itr) -> Tuple
```

Compute both the minimum and maximum of `f` applied to each element in `itr` and return them as a 2-tuple. Only one pass is made over `itr`.

> **Julia 1.2**
>
> This method requires Julia 1.2 or later.

Examples

```
julia> extrema(sin, 0:π)
(0.0, 0.9092974268256817)
```

```
extrema(A::AbstractArray; dims) -> Array{Tuple}
```

Compute the minimum and maximum elements of an array over the given dimensions.

Examples

```
julia> A = reshape(Vector(1:2:16), (2,2,2))
2×2×2 Array{Int64,3}:
[:, :, 1] =
 1  5
 3  7

[:, :, 2] =
  9  13
 11  15

julia> extrema(A, dims = (1,2))
1×1×2 Array{Tuple{Int64,Int64},3}:
[:, :, 1] =
 (1, 7)

[:, :, 2] =
 (9, 15)
```

```
extrema(f, A::AbstractArray; dims) -> Array{Tuple}
```

Compute the minimum and maximum of `f` applied to each element in the given dimensions of `A`.

> **Julia 1.2**
>
> This method requires Julia 1.2 or later.

`Base.argmax` — Function

```
argmax(itr) -> Integer
```

Return the index of the maximum element in a collection. If there are multiple maximal elements, then the first one will be returned.

The collection must not be empty.

Examples

```
julia> argmax([8,0.1,-9,pi])
1

julia> argmax([1,7,7,6])
2

julia> argmax([1,7,7,NaN])
4
```

```
argmax(A; dims) -> indices
```

For an array input, return the indices of the maximum elements over the given dimensions. `NaN` is treated as greater than all other values.

Examples

```
julia> A = [1.0 2; 3 4]
2×2 Array{Float64,2}:
 1.0  2.0
 3.0  4.0
```

```
julia> argmax(A, dims=1)
1×2 Array{CartesianIndex{2},2}:
 CartesianIndex(2, 1)  CartesianIndex(2, 2)

julia> argmax(A, dims=2)
2×1 Array{CartesianIndex{2},2}:
 CartesianIndex(1, 2)
 CartesianIndex(2, 2)
```

`Base.argmin` — Function

```
argmin(itr) -> Integer
```

Return the index of the minimum element in a collection. If there are multiple minimal elements, then the first one will be returned.

The collection must not be empty.

Examples

```
julia> argmin([8,0.1,-9,pi])
3

julia> argmin([7,1,1,6])
2

julia> argmin([7,1,1,NaN])
4
```

```
argmin(A; dims) -> indices
```

For an array input, return the indices of the minimum elements over the given dimensions. `NaN` is treated as less than all other values.

Examples

```
julia> A = [1.0 2; 3 4]
2×2 Array{Float64,2}:
 1.0  2.0
```

```
 3.0  4.0

julia> argmin(A, dims=1)
1×2 Array{CartesianIndex{2},2}:
 CartesianIndex(1, 1)  CartesianIndex(1, 2)

julia> argmin(A, dims=2)
2×1 Array{CartesianIndex{2},2}:
 CartesianIndex(1, 1)
 CartesianIndex(2, 1)
```

`Base.findmax` — Function

```
findmax(itr) -> (x, index)
```

Return the maximum element of the collection `itr` and its index. If there are multiple maximal elements, then the first one will be returned. If any data element is `NaN`, this element is returned. The result is in line with `max`.

The collection must not be empty.

Examples

```
julia> findmax([8,0.1,-9,pi])
(8.0, 1)

julia> findmax([1,7,7,6])
(7, 2)

julia> findmax([1,7,7,NaN])
(NaN, 4)
```

```
findmax(A; dims) -> (maxval, index)
```

For an array input, returns the value and index of the maximum over the given dimensions. `NaN` is treated as greater than all other values.

Examples

```
julia> A = [1.0 2; 3 4]
2×2 Array{Float64,2}:
 1.0  2.0
 3.0  4.0

julia> findmax(A, dims=1)
([3.0 4.0], CartesianIndex{2}[CartesianIndex(2, 1) CartesianIndex(2, 2)])

julia> findmax(A, dims=2)
([2.0; 4.0], CartesianIndex{2}[CartesianIndex(1, 2); CartesianIndex(2, 2)])
```

`Base.findmin` — Function

```
findmin(itr) -> (x, index)
```

Return the minimum element of the collection `itr` and its index. If there are multiple minimal elements, then the first one will be returned. If any data element is `NaN`, this element is returned. The result is in line with `min`.

The collection must not be empty.

Examples

```
julia> findmin([8,0.1,-9,pi])
(-9.0, 3)

julia> findmin([7,1,1,6])
(1, 2)

julia> findmin([7,1,1,NaN])
(NaN, 4)
```

```
findmin(A; dims) -> (minval, index)
```

For an array input, returns the value and index of the minimum over the given dimensions. `NaN` is treated as less than all other values.

Examples

```
julia> A = [1.0 2; 3 4]
2×2 Array{Float64,2}:
 1.0  2.0
 3.0  4.0

julia> findmin(A, dims=1)
([1.0 2.0], CartesianIndex{2}[CartesianIndex(1, 1) CartesianIndex(1, 2)])

julia> findmin(A, dims=2)
([1.0; 3.0], CartesianIndex{2}[CartesianIndex(1, 1); CartesianIndex(2, 1)])
```

`Base.findmax!` — Function

```
findmax!(rval, rind, A) -> (maxval, index)
```

Find the maximum of `A` and the corresponding linear index along singleton dimensions of `rval` and `rind`, and store the results in `rval` and `rind`. `NaN` is treated as greater than all other values.

`Base.findmin!` — Function

```
findmin!(rval, rind, A) -> (minval, index)
```

Find the minimum of `A` and the corresponding linear index along singleton dimensions of `rval` and `rind`, and store the results in `rval` and `rind`. `NaN` is treated as less than all other values.

`Base.sum` — Function

```
sum(f, itr)
```

Sum the results of calling function `f` on each element of `itr`.

The return type is `Int` for signed integers of less than system word size, and `UInt` for unsigned integers of less than system word size. For all other arguments, a common return type is found to which all arguments are promoted.

Examples

```
julia> sum(abs2, [2; 3; 4])
29
```

Note the important difference between `sum(A)` and `reduce(+, A)` for arrays with small integer eltype:

```
julia> sum(Int8[100, 28])
128

julia> reduce(+, Int8[100, 28])
-128
```

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

```
sum(itr)
```

Returns the sum of all elements in a collection.

The return type is `Int` for signed integers of less than system word size, and `UInt` for unsigned integers of less than system word size. For all other arguments, a common return type is found to which all arguments are promoted.

Examples

```
julia> sum(1:20)
210
```

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

Sum elements of an array over the given dimensions.

Examples

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

```
julia> sum(A, dims=1)
1×2 Array{Int64,2}:
 4  6

julia> sum(A, dims=2)
2×1 Array{Int64,2}:
 3
 7
```

Base.sum! — Function

```
sum!(r, A)
```

Sum elements of `A` over the singleton dimensions of `r`, and write results to `r`.

Examples

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

julia> sum!([1; 1], A)
2-element Array{Int64,1}:
 3
 7

julia> sum!([1 1], A)
1×2 Array{Int64,2}:
 4  6
```

Base.prod — Function

```
prod(f, itr)
```

Returns the product of `f` applied to each element of `itr`.

The return type is `Int` for signed integers of less than system word size, and `UInt` for unsigned integers of less than system word size. For all other arguments, a common return type is found to

which all arguments are promoted.

Examples

```
julia> prod(abs2, [2; 3; 4])
576
```

```
prod(itr)
```

Returns the product of all elements of a collection.

The return type is `Int` for signed integers of less than system word size, and `UInt` for unsigned integers of less than system word size. For all other arguments, a common return type is found to which all arguments are promoted.

Examples

```
julia> prod(1:20)
2432902008176640000
```

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

Multiply elements of an array over the given dimensions.

Examples

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

julia> prod(A, dims=1)
1×2 Array{Int64,2}:
 3  8

julia> prod(A, dims=2)
2×1 Array{Int64,2}:
  2
 12
```

`Base.prod!` — Function

```
prod!(r, A)
```

Multiply elements of `A` over the singleton dimensions of `r`, and write results to `r`.

Examples

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

julia> prod!([1; 1], A)
2-element Array{Int64,1}:
  2
 12

julia> prod!([1 1], A)
1×2 Array{Int64,2}:
 3  8
```

`Base.any` — Method

```
any(itr) -> Bool
```

Test whether any elements of a boolean collection are `true`, returning `true` as soon as the first `true` value in `itr` is encountered (short-circuiting).

If the input contains `missing` values, return `missing` if all non-missing values are `false` (or equivalently, if the input contains no `true` value), following three-valued logic.

Examples

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

```
 1

julia> any(a)
true

julia> any((println(i); v) for (i, v) in enumerate(a))
1
true

julia> any([missing, true])
true

julia> any([false, missing])
missing
```

`Base.any` — Method

```
any(p, itr) -> Bool
```

Determine whether predicate `p` returns `true` for any elements of `itr`, returning `true` as soon as the first item in `itr` for which `p` returns `true` is encountered (short-circuiting).

If the input contains `missing` values, return `missing` if all non-missing values are `false` (or equivalently, if the input contains no `true` value), following three-valued logic.

Examples

```
julia> any(i->(4<=i<=6), [3,5,7])
true

julia> any(i -> (println(i); i > 3), 1:10)
1
2
3
4
true

julia> any(i -> i > 0, [1, missing])
true

julia> any(i -> i > 0, [-1, missing])
missing
```

```
julia> any(i -> i > 0, [-1, 0])
false
```

Base.any! — Function

```
any!(r, A)
```

Test whether any values in `A` along the singleton dimensions of `r` are `true`, and write results to `r`.

Examples

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

julia> any!([1; 1], A)
2-element Array{Int64,1}:
 1
 1

julia> any!([1 1], A)
1×2 Array{Int64,2}:
 1  0
```

Base.all — Method

```
all(itr) -> Bool
```

Test whether all elements of a boolean collection are `true`, returning `false` as soon as the first `false` value in `itr` is encountered (short-circuiting).

If the input contains `missing` values, return `missing` if all non-missing values are `true` (or equivalently, if the input contains no `false` value), following three-valued logic.

Examples

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

julia> all(a)
false

julia> all((println(i); v) for (i, v) in enumerate(a))
1
2
false

julia> all([missing, false])
false

julia> all([true, missing])
missing
```

`Base.all` — Method

```
all(p, itr) -> Bool
```

Determine whether predicate `p` returns `true` for all elements of `itr`, returning `false` as soon as the first item in `itr` for which `p` returns `false` is encountered (short-circuiting).

If the input contains `missing` values, return `missing` if all non-missing values are `true` (or equivalently, if the input contains no `false` value), following three-valued logic.

Examples

```
julia> all(i->(4<=i<=6), [4,5,6])
true

julia> all(i -> (println(i); i < 3), 1:10)
1
2
3
false
```

```
julia> all(i -> i > 0, [1, missing])
missing

julia> all(i -> i > 0, [-1, missing])
false

julia> all(i -> i > 0, [1, 2])
true
```

Base.all! — Function

```
all!(r, A)
```

Test whether all values in `A` along the singleton dimensions of `r` are `true`, and write results to `r`.

Examples

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

julia> all!([1; 1], A)
2-element Array{Int64,1}:
 0
 0

julia> all!([1 1], A)
1×2 Array{Int64,2}:
 1  0
```

Base.count — Function

```
count(p, itr) -> Integer
count(itr) -> Integer
```

Count the number of elements in `itr` for which predicate `p` returns `true`. If `p` is omitted, counts

the number of `true` elements in `itr` (which should be a collection of boolean values).

Examples

```
julia> count(i->(4<=i<=6), [2,3,4,5,6])
3

julia> count([true, false, true, true])
3
```

```
count(
    pattern::Union{AbstractString,Regex},
    string::AbstractString;
    overlap::Bool = false,
)
```

Return the number of matches for `pattern` in `string`. This is equivalent to calling `length(findall(pattern, string))` but more efficient.

If `overlap=true`, the matching sequences are allowed to overlap indices in the original string, otherwise they must be from disjoint character ranges.

```
count([f=identity,] A::AbstractArray; dims=:)
```

Count the number of elements in `A` for which `f` returns `true` over the given dimensions.

> **Julia 1.5**
>
> `dims` keyword was added in Julia 1.5.

Examples

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

julia> count(<=(2), A, dims=1)
1×2 Array{Int64,2}:
```

```
 1  1

julia> count(<=(2), A, dims=2)
2×1 Array{Int64,2}:
 2
 0
```

Base.any — Method

```
any(p, itr) -> Bool
```

Determine whether predicate `p` returns `true` for any elements of `itr`, returning `true` as soon as the first item in `itr` for which `p` returns `true` is encountered (short-circuiting).

If the input contains `missing` values, return `missing` if all non-missing values are `false` (or equivalently, if the input contains no `true` value), following three-valued logic.

Examples

```
julia> any(i->(4<=i<=6), [3,5,7])
true

julia> any(i -> (println(i); i > 3), 1:10)
1
2
3
4
true

julia> any(i -> i > 0, [1, missing])
true

julia> any(i -> i > 0, [-1, missing])
missing

julia> any(i -> i > 0, [-1, 0])
false
```

Base.all — Method

```
all(p, itr) -> Bool
```

Determine whether predicate `p` returns `true` for all elements of `itr`, returning `false` as soon as the first item in `itr` for which `p` returns `false` is encountered (short-circuiting).

If the input contains `missing` values, return `missing` if all non-missing values are `true` (or equivalently, if the input contains no `false` value), following three-valued logic.

Examples

```
julia> all(i->(4<=i<=6), [4,5,6])
true

julia> all(i -> (println(i); i < 3), 1:10)
1
2
3
false

julia> all(i -> i > 0, [1, missing])
missing

julia> all(i -> i > 0, [-1, missing])
false

julia> all(i -> i > 0, [1, 2])
true
```

`Base.foreach` — Function

```
foreach(f, c...) -> Nothing
```

Call function `f` on each element of iterable `c`. For multiple iterable arguments, `f` is called elementwise. `foreach` should be used instead of `map` when the results of `f` are not needed, for example in `foreach(println, array)`.

Examples

```
julia> a = 1:3:7;
```

```
julia> foreach(x -> println(x^2), a)
1
16
49
```

`Base.map` — Function

```
map(f, c...) -> collection
```

Transform collection `c` by applying `f` to each element. For multiple collection arguments, apply `f` elementwise.

See also: `mapslices`

Examples

```
julia> map(x -> x * 2, [1, 2, 3])
3-element Array{Int64,1}:
 2
 4
 6

julia> map(+, [1, 2, 3], [10, 20, 30])
3-element Array{Int64,1}:
 11
 22
 33
```

`Base.map!` — Function

```
map!(function, destination, collection...)
```

Like `map`, but stores the result in `destination` rather than a new collection. `destination` must be at least as large as the first collection.

Examples

```
julia> a = zeros(3);

julia> map!(x -> x * 2, a, [1, 2, 3]);

julia> a
3-element Array{Float64,1}:
 2.0
 4.0
 6.0
```

```
map!(f, values(dict::AbstractDict))
```

Modifies `dict` by transforming each value from `val` to `f(val)`. Note that the type of `dict` cannot be changed: if `f(val)` is not an instance of the value type of `dict` then it will be converted to the value type if possible and otherwise raise an error.

> **Julia 1.2**
>
> `map!(f, values(dict::AbstractDict))` requires Julia 1.2 or later.

Examples

```
julia> d = Dict(:a => 1, :b => 2)
Dict{Symbol,Int64} with 2 entries:
  :a => 1
  :b => 2

julia> map!(v -> v-1, values(d))
Base.ValueIterator for a Dict{Symbol,Int64} with 2 entries. Values:
  0
  1
```

`Base.mapreduce` — Method

```
mapreduce(f, op, itrs...; [init])
```

Apply function `f` to each element(s) in `itrs`, and then reduce the result using the binary function

`op`. If provided, `init` must be a neutral element for `op` that will be returned for empty collections. It is unspecified whether `init` is used for non-empty collections. In general, it will be necessary to provide `init` to work with empty collections.

`mapreduce` is functionally equivalent to calling `reduce(op, map(f, itr); init=init)`, but will in general execute faster since no intermediate collection needs to be created. See documentation for `reduce` and `map`.

> **Julia 1.2**
>
> `mapreduce` with multiple iterators requires Julia 1.2 or later.

Examples

```
julia> mapreduce(x->x^2, +, [1:3;]) # == 1 + 4 + 9
14
```

The associativity of the reduction is implementation-dependent. Additionally, some implementations may reuse the return value of `f` for elements that appear multiple times in `itr`. Use `mapfoldl` or `mapfoldr` instead for guaranteed left or right associativity and invocation of `f` for every value.

`Base.mapfoldl` — Method

```
mapfoldl(f, op, itr; [init])
```

Like `mapreduce`, but with guaranteed left associativity, as in `foldl`. If provided, the keyword argument `init` will be used exactly once. In general, it will be necessary to provide `init` to work with empty collections.

`Base.mapfoldr` — Method

```
mapfoldr(f, op, itr; [init])
```

Like `mapreduce`, but with guaranteed right associativity, as in `foldr`. If provided, the keyword argument `init` will be used exactly once. In general, it will be necessary to provide `init` to work

with empty collections.

`Base.first` — Function

```
first(coll)
```

Get the first element of an iterable collection. Return the start point of an `AbstractRange` even if it is empty.

Examples

```
julia> first(2:2:10)
2

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

```
first(s::AbstractString, n::Integer)
```

Get a string consisting of the first `n` characters of `s`.

Examples

```
julia> first("∀ϵ≠0: ϵ²>0", 0)
""

julia> first("∀ϵ≠0: ϵ²>0", 1)
"∀"

julia> first("∀ϵ≠0: ϵ²>0", 3)
"∀ϵ≠"
```

`Base.last` — Function

```
last(coll)
```

Get the last element of an ordered collection, if it can be computed in O(1) time. This is

accomplished by calling `lastindex` to get the last index. Return the end point of an `AbstractRange` even if it is empty.

Examples

```
julia> last(1:2:10)
9

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

```
last(s::AbstractString, n::Integer)
```

Get a string consisting of the last `n` characters of `s`.

Examples

```
julia> last("∀ϵ≠0: ϵ²>0", 0)
""

julia> last("∀ϵ≠0: ϵ²>0", 1)
"0"

julia> last("∀ϵ≠0: ϵ²>0", 3)
"²>0"
```

`Base.front` — Function

```
front(x::Tuple)::Tuple
```

Return a `Tuple` consisting of all but the last component of `x`.

Examples

```
julia> Base.front((1,2,3))
(1, 2)

julia> Base.front(())
ERROR: ArgumentError: Cannot call front on an empty tuple.
```

`Base.tail` — Function

```
tail(x::Tuple)::Tuple
```

Return a `Tuple` consisting of all but the first component of `x`.

Examples

```
julia> Base.tail((1,2,3))
(2, 3)

julia> Base.tail(())
ERROR: ArgumentError: Cannot call tail on an empty tuple.
```

`Base.step` — Function

```
step(r)
```

Get the step size of an `AbstractRange` object.

Examples

```
julia> step(1:10)
1

julia> step(1:2:10)
2

julia> step(2.5:0.3:10.9)
0.3

julia> step(range(2.5, stop=10.9, length=85))
```

```
0.1
```

`Base.collect` — Method

```
collect(collection)
```

Return an `Array` of all items in a collection or iterator. For dictionaries, returns `Pair{KeyType, ValType}`. If the argument is array-like or is an iterator with the `HasShape` trait, the result will have the same shape and number of dimensions as the argument.

Examples

```
julia> collect(1:2:13)
7-element Array{Int64,1}:
  1
  3
  5
  7
  9
 11
 13
```

`Base.collect` — Method

```
collect(element_type, collection)
```

Return an `Array` with the given element type of all items in a collection or iterable. The result has the same shape and number of dimensions as `collection`.

Examples

```
julia> collect(Float64, 1:2:5)
3-element Array{Float64,1}:
 1.0
 3.0
 5.0
```

`Base.filter` — Function

```
filter(f, a)
```

Return a copy of collection `a`, removing elements for which `f` is `false`. The function `f` is passed one argument.

> **Julia 1.4**
>
> Support for `a` as a tuple requires at least Julia 1.4.

Examples

```
julia> a = 1:10
1:10

julia> filter(isodd, a)
5-element Array{Int64,1}:
 1
 3
 5
 7
 9
```

```
filter(f, d::AbstractDict)
```

Return a copy of `d`, removing elements for which `f` is `false`. The function `f` is passed `key=>value` pairs.

Examples

```
julia> d = Dict(1=>"a", 2=>"b")
Dict{Int64,String} with 2 entries:
  2 => "b"
  1 => "a"

julia> filter(p->isodd(p.first), d)
Dict{Int64,String} with 1 entry:
```

```
  1 => "a"
```

```
filter(f, itr::SkipMissing{<:AbstractArray})
```

Return a vector similar to the array wrapped by the given `SkipMissing` iterator but with all missing elements and those for which `f` returns `false` removed.

> **Julia 1.2**
>
> This method requires Julia 1.2 or later.

Examples

```
julia> x = [1 2; missing 4]
2×2 Array{Union{Missing, Int64},2}:
 1         2
  missing  4

julia> filter(isodd, skipmissing(x))
1-element Array{Int64,1}:
 1
```

`Base.filter!` — Function

```
filter!(f, a)
```

Update collection `a`, removing elements for which `f` is `false`. The function `f` is passed one argument.

Examples

```
julia> filter!(isodd, Vector(1:10))
5-element Array{Int64,1}:
 1
 3
 5
 7
```

```
9
```

```
filter!(f, d::AbstractDict)
```

Update `d`, removing elements for which `f` is `false`. The function `f` is passed `key=>value` pairs.

Example

```
julia> d = Dict(1=>"a", 2=>"b", 3=>"c")
Dict{Int64,String} with 3 entries:
  2 => "b"
  3 => "c"
  1 => "a"

julia> filter!(p->isodd(p.first), d)
Dict{Int64,String} with 2 entries:
  3 => "c"
  1 => "a"
```

`Base.replace` — Method

```
replace(A, old_new::Pair...; [count::Integer])
```

Return a copy of collection `A` where, for each pair `old=>new` in `old_new`, all occurrences of `old` are replaced by `new`. Equality is determined using `isequal`. If `count` is specified, then replace at most `count` occurrences in total.

The element type of the result is chosen using promotion (see `promote_type`) based on the element type of `A` and on the types of the `new` values in pairs. If `count` is omitted and the element type of `A` is a `Union`, the element type of the result will not include singleton types which are replaced with values of a different type: for example, `Union{T,Missing}` will become `T` if `missing` is replaced.

See also `replace!`.

Examples

```
julia> replace([1, 2, 1, 3], 1=>0, 2=>4, count=2)
4-element Array{Int64,1}:
```

```
 0
 4
 1
 3

julia> replace([1, missing], missing=>0)
2-element Array{Int64,1}:
 1
 0
```

`Base.replace` — Method

```
replace(new::Function, A; [count::Integer])
```

Return a copy of `A` where each value `x` in `A` is replaced by `new(x)`. If `count` is specified, then replace at most `count` values in total (replacements being defined as `new(x) !== x`).

Examples

```
julia> replace(x -> isodd(x) ? 2x : x, [1, 2, 3, 4])
4-element Array{Int64,1}:
 2
 2
 6
 4

julia> replace(Dict(1=>2, 3=>4)) do kv
           first(kv) < 3 ? first(kv)=>3 : kv
       end
Dict{Int64,Int64} with 2 entries:
  3 => 4
  1 => 3
```

`Base.replace!` — Function

```
replace!(A, old_new::Pair...; [count::Integer])
```

For each pair `old=>new` in `old_new`, replace all occurrences of `old` in collection `A` by `new`. Equality

is determined using `isequal`. If `count` is specified, then replace at most `count` occurrences in total. See also `replace`.

Examples

```
julia> replace!([1, 2, 1, 3], 1=>0, 2=>4, count=2)
4-element Array{Int64,1}:
 0
 4
 1
 3

julia> replace!(Set([1, 2, 3]), 1=>0)
Set{Int64} with 3 elements:
  0
  2
  3
```

```
replace!(new::Function, A; [count::Integer])
```

Replace each element `x` in collection `A` by `new(x)`. If `count` is specified, then replace at most `count` values in total (replacements being defined as `new(x) !== x`).

Examples

```
julia> replace!(x -> isodd(x) ? 2x : x, [1, 2, 3, 4])
4-element Array{Int64,1}:
 2
 2
 6
 4

julia> replace!(Dict(1=>2, 3=>4)) do kv
           first(kv) < 3 ? first(kv)=>3 : kv
       end
Dict{Int64,Int64} with 2 entries:
  3 => 4
  1 => 3

julia> replace!(x->2x, Set([3, 6]))
Set{Int64} with 2 elements:
  6
```

```
12
```

Indexable Collections

`Base.getindex` — Function

```
getindex(collection, key...)
```

Retrieve the value(s) stored at the given key or index within a collection. The syntax `a[i,j,...]` is converted by the compiler to `getindex(a, i, j, ...)`.

Examples

```
julia> A = Dict("a" => 1, "b" => 2)
Dict{String,Int64} with 2 entries:
  "b" => 2
  "a" => 1

julia> getindex(A, "a")
1
```

`Base.setindex!` — Function

```
setindex!(collection, value, key...)
```

Store the given value at the given key or index within a collection. The syntax `a[i,j,...] = x` is converted by the compiler to `(setindex!(a, x, i, j, ...); x)`.

`Base.firstindex` — Function

```
firstindex(collection) -> Integer
firstindex(collection, d) -> Integer
```

Return the first index of `collection`. If `d` is given, return the first index of `collection` along dimension `d`.

Examples

```
julia> firstindex([1,2,4])
1

julia> firstindex(rand(3,4,5), 2)
1
```

`Base.lastindex` — Function

```
lastindex(collection) -> Integer
lastindex(collection, d) -> Integer
```

Return the last index of `collection`. If `d` is given, return the last index of `collection` along dimension `d`.

The syntaxes `A[end]` and `A[end, end]` lower to `A[lastindex(A)]` and `A[lastindex(A, 1), lastindex(A, 2)]`, respectively.

Examples

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

julia> lastindex(rand(3,4,5), 2)
4
```

Fully implemented by:

- `Array`
- `BitArray`
- `AbstractArray`
- `SubArray`

Partially implemented by:

- `AbstractRange`
- `UnitRange`
- `Tuple`

- AbstractString
- Dict
- IdDict
- WeakKeyDict
- NamedTuple

Dictionaries

Dict is the standard dictionary. Its implementation uses hash as the hashing function for the key, and isequal to determine equality. Define these two functions for custom types to override how they are stored in a hash table.

IdDict is a special hash table where the keys are always object identities.

WeakKeyDict is a hash table implementation where the keys are weak references to objects, and thus may be garbage collected even when referenced in a hash table. Like Dict it uses hash for hashing and isequal for equality, unlike Dict it does not convert keys on insertion.

Dicts can be created by passing pair objects constructed with => to a Dict constructor: Dict("A"=>1, "B"=>2). This call will attempt to infer type information from the keys and values (i.e. this example creates a Dict{String, Int64}). To explicitly specify types use the syntax Dict{KeyType,ValueType}(...). For example, Dict{String,Int32}("A"=>1, "B"=>2).

Dictionaries may also be created with generators. For example, Dict(i => f(i) for i = 1:10).

Given a dictionary D, the syntax D[x] returns the value of key x (if it exists) or throws an error, and D[x] = y stores the key-value pair x => y in D (replacing any existing value for the key x). Multiple arguments to D[...] are converted to tuples; for example, the syntax D[x,y] is equivalent to D[(x,y)], i.e. it refers to the value keyed by the tuple (x,y).

Base.AbstractDict — Type

```
AbstractDict{K, V}
```

Supertype for dictionary-like types with keys of type K and values of type V. Dict, IdDict and other types are subtypes of this. An AbstractDict{K, V} should be an iterator of Pair{K, V}.

Base.Dict — Type

```
Dict([itr])
```

`Dict{K,V}()` constructs a hash table with keys of type `K` and values of type `V`. Keys are compared with `isequal` and hashed with `hash`.

Given a single iterable argument, constructs a `Dict` whose key-value pairs are taken from 2-tuples `(key,value)` generated by the argument.

Examples

```
julia> Dict([("A", 1), ("B", 2)])
Dict{String,Int64} with 2 entries:
  "B" => 2
  "A" => 1
```

Alternatively, a sequence of pair arguments may be passed.

```
julia> Dict("A"=>1, "B"=>2)
Dict{String,Int64} with 2 entries:
  "B" => 2
  "A" => 1
```

`Base.IdDict` — Type

```
IdDict([itr])
```

`IdDict{K,V}()` constructs a hash table using object-id as hash and `===` as equality with keys of type `K` and values of type `V`.

See `Dict` for further help.

`Base.WeakKeyDict` — Type

```
WeakKeyDict([itr])
```

`WeakKeyDict()` constructs a hash table where the keys are weak references to objects which may

be garbage collected even when referenced in a hash table.

See `Dict` for further help. Note, unlike `Dict`, `WeakKeyDict` does not convert keys on insertion.

`Base.ImmutableDict` — Type

```
ImmutableDict
```

`ImmutableDict` is a dictionary implemented as an immutable linked list, which is optimal for small dictionaries that are constructed over many individual insertions. Note that it is not possible to remove a value, although it can be partially overridden and hidden by inserting a new value with the same key.

```
ImmutableDict(KV::Pair)
```

Create a new entry in the `ImmutableDict` for a `key => value` pair

- use `(key => value) in dict` to see if this particular combination is in the properties set
- use `get(dict, key, default)` to retrieve the most recent value for a particular key

`Base.haskey` — Function

```
haskey(collection, key) -> Bool
```

Determine whether a collection has a mapping for a given `key`.

Examples

```
julia> D = Dict('a'=>2, 'b'=>3)
Dict{Char,Int64} with 2 entries:
  'a' => 2
  'b' => 3

julia> haskey(D, 'a')
true

julia> haskey(D, 'c')
false
```

`Base.get` — Method

```
get(collection, key, default)
```

Return the value stored for the given key, or the given default value if no mapping for the key is present.

Examples

```
julia> d = Dict("a"=>1, "b"=>2);

julia> get(d, "a", 3)
1

julia> get(d, "c", 3)
3
```

`Base.get` — Function

```
get(collection, key, default)
```

Return the value stored for the given key, or the given default value if no mapping for the key is present.

Examples

```
julia> d = Dict("a"=>1, "b"=>2);

julia> get(d, "a", 3)
1

julia> get(d, "c", 3)
3
```

```
get(f::Function, collection, key)
```

Return the value stored for the given key, or if no mapping for the key is present, return `f()`. Use `get!` to also store the default value in the dictionary.

This is intended to be called using `do` block syntax

```
get(dict, key) do
    # default value calculated here
    time()
end
```

`Base.get!` — Method

```
get!(collection, key, default)
```

Return the value stored for the given key, or if no mapping for the key is present, store `key => default`, and return `default`.

Examples

```
julia> d = Dict("a"=>1, "b"=>2, "c"=>3);
```

```
julia> get!(d, "a", 5)
1

julia> get!(d, "d", 4)
4

julia> d
Dict{String,Int64} with 4 entries:
  "c" => 3
  "b" => 2
  "a" => 1
  "d" => 4
```

`Base.get!` — Method

```
get!(f::Function, collection, key)
```

Return the value stored for the given key, or if no mapping for the key is present, store `key => f()`, and return `f()`.

This is intended to be called using `do` block syntax:

```
get!(dict, key) do
    # default value calculated here
    time()
end
```

`Base.getkey` — Function

```
getkey(collection, key, default)
```

Return the key matching argument `key` if one exists in `collection`, otherwise return `default`.

Examples

```
julia> D = Dict('a'=>2, 'b'=>3)
Dict{Char,Int64} with 2 entries:
```

```
  'a' => 2
  'b' => 3

julia> getkey(D, 'a', 1)
'a': ASCII/Unicode U+0061 (category Ll: Letter, lowercase)

julia> getkey(D, 'd', 'a')
'a': ASCII/Unicode U+0061 (category Ll: Letter, lowercase)
```

`Base.delete!` — Function

```
delete!(collection, key)
```

Delete the mapping for the given key in a collection, if any, and return the collection.

Examples

```
julia> d = Dict("a"=>1, "b"=>2)
Dict{String,Int64} with 2 entries:
  "b" => 2
  "a" => 1

julia> delete!(d, "b")
Dict{String,Int64} with 1 entry:
  "a" => 1

julia> delete!(d, "b") # d is left unchanged
Dict{String,Int64} with 1 entry:
  "a" => 1
```

`Base.pop!` — Method

```
pop!(collection, key[, default])
```

Delete and return the mapping for `key` if it exists in `collection`, otherwise return `default`, or throw an error if `default` is not specified.

Examples

```
julia> d = Dict("a"=>1, "b"=>2, "c"=>3);

julia> pop!(d, "a")
1

julia> pop!(d, "d")
ERROR: KeyError: key "d" not found
Stacktrace:
[...]

julia> pop!(d, "e", 4)
4
```

Base.keys — Function

```
keys(iterator)
```

For an iterator or collection that has keys and values (e.g. arrays and dictionaries), return an iterator over the keys.

Base.values — Function

```
values(iterator)
```

For an iterator or collection that has keys and values, return an iterator over the values. This function simply returns its argument by default, since the elements of a general iterator are normally considered its "values".

Examples

```
julia> d = Dict("a"=>1, "b"=>2);

julia> values(d)
Base.ValueIterator for a Dict{String,Int64} with 2 entries. Values:
  2
  1

julia> values([2])
```

```
1-element Array{Int64,1}:
 2
```

```
values(a::AbstractDict)
```

Return an iterator over all values in a collection. `collect(values(a))` returns an array of values. When the values are stored internally in a hash table, as is the case for `Dict`, the order in which they are returned may vary. But `keys(a)` and `values(a)` both iterate `a` and return the elements in the same order.

Examples

```
julia> D = Dict('a'=>2, 'b'=>3)
Dict{Char,Int64} with 2 entries:
  'a' => 2
  'b' => 3

julia> collect(values(D))
2-element Array{Int64,1}:
 2
 3
```

`Base.pairs` — Function

```
pairs(IndexLinear(), A)
pairs(IndexCartesian(), A)
pairs(IndexStyle(A), A)
```

An iterator that accesses each element of the array `A`, returning `i => x`, where `i` is the index for the element and `x = A[i]`. Identical to `pairs(A)`, except that the style of index can be selected. Also similar to `enumerate(A)`, except `i` will be a valid index for `A`, while `enumerate` always counts from 1 regardless of the indices of `A`.

Specifying `IndexLinear()` ensures that `i` will be an integer; specifying `IndexCartesian()` ensures that `i` will be a `CartesianIndex`; specifying `IndexStyle(A)` chooses whichever has been defined as the native indexing style for array `A`.

Mutation of the bounds of the underlying array will invalidate this iterator.

Examples

```
julia> A = ["a" "d"; "b" "e"; "c" "f"];

julia> for (index, value) in pairs(IndexStyle(A), A)
           println("$index $value")
       end
1 a
2 b
3 c
4 d
5 e
6 f

julia> S = view(A, 1:2, :);

julia> for (index, value) in pairs(IndexStyle(S), S)
           println("$index $value")
       end
CartesianIndex(1, 1) a
CartesianIndex(2, 1) b
CartesianIndex(1, 2) d
CartesianIndex(2, 2) e
```

See also: `IndexStyle`, `axes`.

```
pairs(collection)
```

Return an iterator over `key => value` pairs for any collection that maps a set of keys to a set of values. This includes arrays, where the keys are the array indices.

`Base.merge` — Function

```
merge(d::AbstractDict, others::AbstractDict...)
```

Construct a merged collection from the given collections. If necessary, the types of the resulting collection will be promoted to accommodate the types of the merged collections. If the same key is present in another collection, the value for that key will be the value it has in the last collection listed.

Examples

```
julia> a = Dict("foo" => 0.0, "bar" => 42.0)
Dict{String,Float64} with 2 entries:
  "bar" => 42.0
  "foo" => 0.0

julia> b = Dict("baz" => 17, "bar" => 4711)
Dict{String,Int64} with 2 entries:
  "bar" => 4711
  "baz" => 17

julia> merge(a, b)
Dict{String,Float64} with 3 entries:
  "bar" => 4711.0
  "baz" => 17.0
  "foo" => 0.0

julia> merge(b, a)
Dict{String,Float64} with 3 entries:
  "bar" => 42.0
  "baz" => 17.0
  "foo" => 0.0
```

```
merge(a::NamedTuple, bs::NamedTuple...)
```

Construct a new named tuple by merging two or more existing ones, in a left-associative manner. Merging proceeds left-to-right, between pairs of named tuples, and so the order of fields present in both the leftmost and rightmost named tuples take the same position as they are found in the leftmost named tuple. However, values are taken from matching fields in the rightmost named tuple that contains that field. Fields present in only the rightmost named tuple of a pair are appended at the end. A fallback is implemented for when only a single named tuple is supplied, with signature `merge(a::NamedTuple)`.

> **Julia 1.1**
>
> Merging 3 or more `NamedTuple` requires at least Julia 1.1.

Examples

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

```
julia> merge((a=1, b=2), (b=3, c=(d=1,)), (c=(d=2,),))
(a = 1, b = 3, c = (d = 2,))
```

```
merge(a::NamedTuple, iterable)
```

Interpret an iterable of key-value pairs as a named tuple, and perform a merge.

```
julia> merge((a=1, b=2, c=3), [:b=>4, :d=>5])
(a = 1, b = 4, c = 3, d = 5)
```

`Base.mergewith` — Function

```
mergewith(combine, d::AbstractDict, others::AbstractDict...)
mergewith(combine)
merge(combine, d::AbstractDict, others::AbstractDict...)
```

Construct a merged collection from the given collections. If necessary, the types of the resulting collection will be promoted to accommodate the types of the merged collections. Values with the same key will be combined using the combiner function. The curried form `mergewith(combine)` returns the function `(args...) -> mergewith(combine, args...)`.

Method `merge(combine::Union{Function,Type}, args...)` as an alias of `mergewith(combine, args...)` is still available for backward compatibility.

Julia 1.5

`mergewith` requires Julia 1.5 or later.

Examples

```
julia> a = Dict("foo" => 0.0, "bar" => 42.0)
Dict{String,Float64} with 2 entries:
  "bar" => 42.0
```

```
  "foo" => 0.0

julia> b = Dict("baz" => 17, "bar" => 4711)
Dict{String,Int64} with 2 entries:
  "bar" => 4711
  "baz" => 17

julia> mergewith(+, a, b)
Dict{String,Float64} with 3 entries:
  "bar" => 4753.0
  "baz" => 17.0
  "foo" => 0.0

julia> ans == mergewith(+)(a, b)
true
```

`Base.merge!` — Function

```
merge!(d::AbstractDict, others::AbstractDict...)
```

Update collection with pairs from the other collections. See also `merge`.

Examples

```
julia> d1 = Dict(1 => 2, 3 => 4);

julia> d2 = Dict(1 => 4, 4 => 5);

julia> merge!(d1, d2);

julia> d1
Dict{Int64,Int64} with 3 entries:
  4 => 5
  3 => 4
  1 => 4
```

`Base.mergewith!` — Function

```
mergewith!(combine, d::AbstractDict, others::AbstractDict...) -> d
```

```
mergewith!(combine)
merge!(combine, d::AbstractDict, others::AbstractDict...) -> d
```

Update collection with pairs from the other collections. Values with the same key will be combined using the combiner function. The curried form `mergewith!(combine)` returns the function `(args...) -> mergewith!(combine, args...)`.

Method `merge!(combine::Union{Function,Type}, args...)` as an alias of `mergewith!(combine, args...)` is still available for backward compatibility.

> **Julia 1.5**
>
> `mergewith!` requires Julia 1.5 or later.

Examples

```
julia> d1 = Dict(1 => 2, 3 => 4);

julia> d2 = Dict(1 => 4, 4 => 5);

julia> mergewith!(+, d1, d2);

julia> d1
Dict{Int64,Int64} with 3 entries:
  4 => 5
  3 => 4
  1 => 6

julia> mergewith!(-, d1, d1);

julia> d1
Dict{Int64,Int64} with 3 entries:
  4 => 0
  3 => 0
  1 => 0

julia> foldl(mergewith!(+), [d1, d2]; init=Dict{Int64,Int64}())
Dict{Int64,Int64} with 3 entries:
  4 => 5
  3 => 0
  1 => 4
```

`Base.sizehint!` — Function

```
sizehint!(s, n)
```

Suggest that collection `s` reserve capacity for at least `n` elements. This can improve performance.

`Base.keytype` — Function

```
keytype(T::Type{<:AbstractArray})
keytype(A::AbstractArray)
```

Return the key type of an array. This is equal to the `eltype` of the result of `keys(...)`, and is provided mainly for compatibility with the dictionary interface.

Examples

```
julia> keytype([1, 2, 3]) == Int
true

julia> keytype([1 2; 3 4])
CartesianIndex{2}
```

> **Julia 1.2**
>
> For arrays, this function requires at least Julia 1.2.

```
keytype(type)
```

Get the key type of an dictionary type. Behaves similarly to `eltype`.

Examples

```
julia> keytype(Dict(Int32(1) => "foo"))
Int32
```

`Base.valtype` — Function

```
valtype(T::Type{<:AbstractArray})
valtype(A::AbstractArray)
```

Return the value type of an array. This is identical to `eltype` and is provided mainly for compatibility with the dictionary interface.

Examples

```
julia> valtype(["one", "two", "three"])
String
```

> **Julia 1.2**
>
> For arrays, this function requires at least Julia 1.2.

```
valtype(type)
```

Get the value type of an dictionary type. Behaves similarly to `eltype`.

Examples

```
julia> valtype(Dict(Int32(1) => "foo"))
String
```

Fully implemented by:

- `IdDict`
- `Dict`
- `WeakKeyDict`

Partially implemented by:

- `BitSet`
- `Set`
- `EnvDict`

- `Array`
- `BitArray`
- `ImmutableDict`
- `Iterators.Pairs`

Set-Like Collections

`Base.AbstractSet` — Type

```
AbstractSet{T}
```

Supertype for set-like types whose elements are of type `T`. `Set`, `BitSet` and other types are subtypes of this.

`Base.Set` — Type

```
Set([itr])
```

Construct a `Set` of the values generated by the given iterable object, or an empty set. Should be used instead of `BitSet` for sparse integer sets, or for sets of arbitrary objects.

`Base.BitSet` — Type

```
BitSet([itr])
```

Construct a sorted set of `Int`s generated by the given iterable object, or an empty set. Implemented as a bit string, and therefore designed for dense integer sets. If the set will be sparse (for example, holding a few very large integers), use `Set` instead.

`Base.union` — Function

```
union(s, itrs...)
∪(s, itrs...)
```

Construct the union of sets. Maintain order with arrays.

Examples

```
julia> union([1, 2], [3, 4])
4-element Array{Int64,1}:
 1
 2
 3
 4

julia> union([1, 2], [2, 4])
3-element Array{Int64,1}:
 1
 2
 4

julia> union([4, 2], 1:2)
3-element Array{Int64,1}:
 4
 2
 1

julia> union(Set([1, 2]), 2:3)
Set{Int64} with 3 elements:
  2
  3
  1
```

`Base.union!` — Function

```
union!(s::Union{AbstractSet,AbstractVector}, itrs...)
```

Construct the union of passed in sets and overwrite `s` with the result. Maintain order with arrays.

Examples

```
julia> a = Set([1, 3, 4, 5]);

julia> union!(a, 1:2:8);
```

```
julia> a
Set{Int64} with 5 elements:
  7
  4
  3
  5
  1
```

Base.intersect — Function

```
intersect(s, itrs...)
∩(s, itrs...)
```

Construct the intersection of sets. Maintain order with arrays.

Examples

```
julia> intersect([1, 2, 3], [3, 4, 5])
1-element Array{Int64,1}:
 3

julia> intersect([1, 4, 4, 5, 6], [4, 6, 6, 7, 8])
2-element Array{Int64,1}:
 4
 6

julia> intersect(Set([1, 2]), BitSet([2, 3]))
Set{Int64} with 1 element:
  2
```

Base.setdiff — Function

```
setdiff(s, itrs...)
```

Construct the set of elements in `s` but not in any of the iterables in `itrs`. Maintain order with arrays.

Examples

```
julia> setdiff([1,2,3], [3,4,5])
2-element Array{Int64,1}:
 1
 2
```

Base.setdiff! — Function

```
setdiff!(s, itrs...)
```

Remove from set `s` (in-place) each element of each iterable from `itrs`. Maintain order with arrays.

Examples

```
julia> a = Set([1, 3, 4, 5]);

julia> setdiff!(a, 1:2:6);

julia> a
Set{Int64} with 1 element:
  4
```

Base.symdiff — Function

```
symdiff(s, itrs...)
```

Construct the symmetric difference of elements in the passed in sets. When `s` is not an `AbstractSet`, the order is maintained. Note that in this case the multiplicity of elements matters.

Examples

```
julia> symdiff([1,2,3], [3,4,5], [4,5,6])
3-element Array{Int64,1}:
 1
 2
 6

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

```
2-element Array{Int64,1}:
 1
 2

julia> symdiff(unique([1,2,1]), unique([2, 1, 2]))
Int64[]
```

Base.symdiff! — Function

```
symdiff!(s::Union{AbstractSet,AbstractVector}, itrs...)
```

Construct the symmetric difference of the passed in sets, and overwrite `s` with the result. When `s` is an array, the order is maintained. Note that in this case the multiplicity of elements matters.

Base.intersect! — Function

```
intersect!(s::Union{AbstractSet,AbstractVector}, itrs...)
```

Intersect all passed in sets and overwrite `s` with the result. Maintain order with arrays.

Base.issubset — Function

```
issubset(a, b) -> Bool
⊆(a, b) -> Bool
⊇(b, a) -> Bool
```

Determine whether every element of `a` is also in `b`, using `in`.

Examples

```
julia> issubset([1, 2], [1, 2, 3])
true

julia> [1, 2, 3] ⊆ [1, 2]
false
```

```
julia> [1, 2, 3] ⊇ [1, 2]
true
```

Base.:⊈ — Function

```
⊈(a, b) -> Bool
⊉(b, a) -> Bool
```

Negation of ⊆ and ⊇, i.e. checks that `a` is not a subset of `b`.

Examples

```
julia> (1, 2) ⊈ (2, 3)
true

julia> (1, 2) ⊈ (1, 2, 3)
false
```

Base.:⊊ — Function

```
⊊(a, b) -> Bool
⊋(b, a) -> Bool
```

Determines if `a` is a subset of, but not equal to, `b`.

Examples

```
julia> (1, 2) ⊊ (1, 2, 3)
true

julia> (1, 2) ⊊ (1, 2)
false
```

Base.issetequal — Function

```
issetequal(a, b) -> Bool
```

Determine whether `a` and `b` have the same elements. Equivalent to `a ⊆ b && b ⊆ a` but more efficient when possible.

Examples

```
julia> issetequal([1, 2], [1, 2, 3])
false

julia> issetequal([1, 2], [2, 1])
true
```

`Base.isdisjoint` — Function

```
isdisjoint(v1, v2) -> Bool
```

Return whether the collections `v1` and `v2` are disjoint, i.e. whether their intersection is empty.

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

Fully implemented by:

- `BitSet`
- `Set`

Partially implemented by:

- `Array`

Dequeues

`Base.push!` — Function

```
push!(collection, items...) -> collection
```

Insert one or more `items` in `collection`. If `collection` is an ordered container, the items are inserted at the end (in the given order).

Examples

```
julia> push!([1, 2, 3], 4, 5, 6)
6-element Array{Int64,1}:
 1
 2
 3
 4
 5
 6
```

If `collection` is ordered, use `append!` to add all the elements of another collection to it. The result of the preceding example is equivalent to `append!([1, 2, 3], [4, 5, 6])`. For `AbstractSet` objects, `union!` can be used instead.

`Base.pop!` — Function

```
pop!(collection) -> item
```

Remove an item in `collection` and return it. If `collection` is an ordered container, the last item is returned.

Examples

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

julia> pop!(A)
3

julia> A
```

```
2-element Array{Int64,1}:
 1
 2

julia> S = Set([1, 2])
Set{Int64} with 2 elements:
  2
  1

julia> pop!(S)
2

julia> S
Set{Int64} with 1 element:
  1

julia> pop!(Dict(1=>2))
1 => 2
```

```
pop!(collection, key[, default])
```

Delete and return the mapping for `key` if it exists in `collection`, otherwise return `default`, or throw an error if `default` is not specified.

Examples

```
julia> d = Dict("a"=>1, "b"=>2, "c"=>3);

julia> pop!(d, "a")
1

julia> pop!(d, "d")
ERROR: KeyError: key "d" not found
Stacktrace:
[...]

julia> pop!(d, "e", 4)
4
```

`Base.popat!` — Function

```
popat!(a::Vector, i::Integer, [default])
```

Remove the item at the given `i` and return it. Subsequent items are shifted to fill the resulting gap. When `i` is not a valid index for `a`, return `default`, or throw an error if `default` is not specified. See also `deleteat!` and `splice!`.

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

Examples

```
julia> a = [4, 3, 2, 1]; popat!(a, 2)
3

julia> a
3-element Array{Int64,1}:
 4
 2
 1

julia> popat!(a, 4, missing)
missing

julia> popat!(a, 4)
ERROR: BoundsError: attempt to access 3-element Array{Int64,1} at index [4]
[...]
```

`Base.pushfirst!` — Function

```
pushfirst!(collection, items...) -> collection
```

Insert one or more `items` at the beginning of `collection`.

Examples

```
julia> pushfirst!([1, 2, 3, 4], 5, 6)
6-element Array{Int64,1}:
```

```
 5
 6
 1
 2
 3
 4
```

Base.popfirst! — Function

```
popfirst!(collection) -> item
```

Remove the first `item` from `collection`.

Examples

```
julia> A = [1, 2, 3, 4, 5, 6]
6-element Array{Int64,1}:
 1
 2
 3
 4
 5
 6

julia> popfirst!(A)
1

julia> A
5-element Array{Int64,1}:
 2
 3
 4
 5
 6
```

Base.insert! — Function

```
insert!(a::Vector, index::Integer, item)
```

Insert an `item` into `a` at the given `index`. `index` is the index of `item` in the resulting `a`.

Examples

```
julia> insert!([6, 5, 4, 2, 1], 4, 3)
6-element Array{Int64,1}:
 6
 5
 4
 3
 2
 1
```

`Base.deleteat!` — Function

```
deleteat!(a::Vector, i::Integer)
```

Remove the item at the given `i` and return the modified `a`. Subsequent items are shifted to fill the resulting gap.

Examples

```
julia> deleteat!([6, 5, 4, 3, 2, 1], 2)
5-element Array{Int64,1}:
 6
 4
 3
 2
 1
```

```
deleteat!(a::Vector, inds)
```

Remove the items at the indices given by `inds`, and return the modified `a`. Subsequent items are shifted to fill the resulting gap.

`inds` can be either an iterator or a collection of sorted and unique integer indices, or a boolean vector of the same length as `a` with `true` indicating entries to delete.

Examples

```
julia> deleteat!([6, 5, 4, 3, 2, 1], 1:2:5)
3-element Array{Int64,1}:
 5
 3
 1

julia> deleteat!([6, 5, 4, 3, 2, 1], [true, false, true, false, true, false])
3-element Array{Int64,1}:
 5
 3
 1

julia> deleteat!([6, 5, 4, 3, 2, 1], (2, 2))
ERROR: ArgumentError: indices must be unique and sorted
Stacktrace:
[...]
```

Base.splice! — Function

```
splice!(a::Vector, index::Integer, [replacement]) -> item
```

Remove the item at the given index, and return the removed item. Subsequent items are shifted left to fill the resulting gap. If specified, replacement values from an ordered collection will be spliced in place of the removed item.

Examples

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

julia> A
5-element Array{Int64,1}:
 6
 5
 4
 3
 1

julia> splice!(A, 5, -1)
1
```

```
julia> A
5-element Array{Int64,1}:
  6
  5
  4
  3
 -1

julia> splice!(A, 1, [-1, -2, -3])
6

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

To insert `replacement` before an index `n` without removing any items, use `splice!(collection, n:n-1, replacement)`.

```
splice!(a::Vector, indices, [replacement]) -> items
```

Remove items at specified indices, and return a collection containing the removed items. Subsequent items are shifted left to fill the resulting gaps. If specified, replacement values from an ordered collection will be spliced in place of the removed items; in this case, `indices` must be a `UnitRange`.

To insert `replacement` before an index `n` without removing any items, use `splice!(collection, n:n-1, replacement)`.

> **Julia 1.5**
>
> Prior to Julia 1.5, `indices` must always be a `UnitRange`.

Examples

```
julia> A = [-1, -2, -3, 5, 4, 3, -1]; splice!(A, 4:3, 2)
```

```
Int64[]

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

`Base.resize!` — Function

```
resize!(a::Vector, n::Integer) -> Vector
```

Resize `a` to contain `n` elements. If `n` is smaller than the current collection length, the first `n` elements will be retained. If `n` is larger, the new elements are not guaranteed to be initialized.

Examples

```
julia> resize!([6, 5, 4, 3, 2, 1], 3)
3-element Array{Int64,1}:
 6
 5
 4

julia> a = resize!([6, 5, 4, 3, 2, 1], 8);

julia> length(a)
8

julia> a[1:6]
6-element Array{Int64,1}:
 6
 5
 4
 3
 2
 1
```

`Base.append!` — Function

```
append!(collection, collection2) -> collection.
```

For an ordered container `collection`, add the elements of `collection2` to the end of it.

Examples

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

```
julia> append!([1, 2, 3], [4, 5, 6])
6-element Array{Int64,1}:
 1
 2
 3
 4
 5
 6
```

Use `push!` to add individual items to `collection` which are not already themselves in another collection. The result of the preceding example is equivalent to `push!([1, 2, 3], 4, 5, 6)`.

`Base.prepend!` — Function

```
prepend!(a::Vector, items) -> collection
```

Insert the elements of `items` to the beginning of `a`.

Examples

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

Fully implemented by:

- `Vector` (a.k.a. 1-dimensional `Array`)
- `BitVector` (a.k.a. 1-dimensional `BitArray`)

Utility Collections

`Base.Pair` — Type

```
Pair(x, y)
x => y
```

Construct a `Pair` object with type `Pair{typeof(x), typeof(y)}`. The elements are stored in the fields `first` and `second`. They can also be accessed via iteration (but a `Pair` is treated as a single "scalar" for broadcasting operations).

See also: `Dict`

Examples

```
julia> p = "foo" => 7
"foo" => 7

julia> typeof(p)
Pair{String,Int64}

julia> p.first
"foo"

julia> for x in p
           println(x)
       end
foo
7
```

`Base.Iterators.Pairs` — Type

```
Iterators.Pairs(values, keys) <: AbstractDict{eltype(keys), eltype(values)}
```

Transforms an indexable container into an Dictionary-view of the same data. Modifying the key-space of the underlying data may invalidate this object.

« Essentials Mathematics »