Iteration utilities

`Base.Iterators.Stateful` — Type

```
Stateful(itr)
```

There are several different ways to think about this iterator wrapper:

1. It provides a mutable wrapper around an iterator and its iteration state.
2. It turns an iterator-like abstraction into a `Channel`-like abstraction.
3. It's an iterator that mutates to become its own rest iterator whenever an item is produced.

`Stateful` provides the regular iterator interface. Like other mutable iterators (e.g. `Channel`), if iteration is stopped early (e.g. by a `break` in a `for` loop), iteration can be resumed from the same spot by continuing to iterate over the same iterator object (in contrast, an immutable iterator would restart from the beginning).

Examples

```
julia> a = Iterators.Stateful("abcdef");

julia> isempty(a)
false

julia> popfirst!(a)
'a': ASCII/Unicode U+0061 (category Ll: Letter, lowercase)

julia> collect(Iterators.take(a, 3))
3-element Array{Char,1}:
 '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)

julia> collect(a)
2-element Array{Char,1}:
 'e': ASCII/Unicode U+0065 (category Ll: Letter, lowercase)
 'f': ASCII/Unicode U+0066 (category Ll: Letter, lowercase)
```

```
julia> a = Iterators.Stateful([1,1,1,2,3,4]);

julia> for x in a; x == 1 || break; end

julia> peek(a)
3

julia> sum(a) # Sum the remaining elements
7
```

`Base.Iterators.zip` — Function

```
zip(iters...)
```

Run multiple iterators at the same time, until any of them is exhausted. The value type of the `zip` iterator is a tuple of values of its subiterators.

Note

`zip` orders the calls to its subiterators in such a way that stateful iterators will not advance when another iterator finishes in the current iteration.

Examples

```
julia> a = 1:5
1:5

julia> b = ["e","d","b","c","a"]
5-element Array{String,1}:
 "e"
 "d"
 "b"
 "c"
 "a"

julia> c = zip(a,b)
zip(1:5, ["e", "d", "b", "c", "a"])

julia> length(c)
```

```
5

julia> first(c)
(1, "e")
```

Base.Iterators.enumerate — Function

```
enumerate(iter)
```

An iterator that yields `(i, x)` where `i` is a counter starting at 1, and `x` is the `i`th value from the given iterator. It's useful when you need not only the values `x` over which you are iterating, but also the number of iterations so far. Note that `i` may not be valid for indexing `iter`; it's also possible that `x != iter[i]`, if `iter` has indices that do not start at 1. See the `pairs(IndexLinear(), iter)` method if you want to ensure that `i` is an index.

Examples

```
julia> a = ["a", "b", "c"];

julia> for (index, value) in enumerate(a)
           println("$index $value")
       end
1 a
2 b
3 c
```

Base.Iterators.rest — Function

```
rest(iter, state)
```

An iterator that yields the same elements as `iter`, but starting at the given `state`.

Examples

```
julia> collect(Iterators.rest([1,2,3,4], 2))
3-element Array{Int64,1}:
 2
 3
 4
```

`Base.Iterators.countfrom` — Function

```
countfrom(start=1, step=1)
```

An iterator that counts forever, starting at `start` and incrementing by `step`.

Examples

```
julia> for v in Iterators.countfrom(5, 2)
           v > 10 && break
           println(v)
       end
5
7
9
```

`Base.Iterators.take` — Function

```
take(iter, n)
```

An iterator that generates at most the first `n` elements of `iter`.

Examples

```
julia> a = 1:2:11
1:2:11

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

```
julia> collect(Iterators.take(a,3))
3-element Array{Int64,1}:
 1
 3
 5
```

`Base.Iterators.takewhile` — Function

```
takewhile(pred, iter)
```

An iterator that generates element from `iter` as long as predicate `pred` is true, afterwards, drops every element.

Julia 1.4

This function requires at least Julia 1.4.

Examples

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

julia> collect(Iterators.takewhile(<(3),s))
2-element Array{Int64,1}:
 1
 2
```

`Base.Iterators.drop` — Function

```
drop(iter, n)
```

An iterator that generates all but the first `n` elements of `iter`.

Examples

```
julia> a = 1:2:11
1:2:11

julia> collect(a)
6-element Array{Int64,1}:
  1
  3
  5
  7
  9
 11

julia> collect(Iterators.drop(a,4))
2-element Array{Int64,1}:
  9
 11
```

`Base.Iterators.dropwhile` — Function

```
dropwhile(pred, iter)
```

An iterator that drops element from `iter` as long as predicate `pred` is true, afterwards, returns every element.

Julia 1.4

This function requires at least Julia 1.4.

Examples

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

```
julia> collect(Iterators.dropwhile(<(3),s))
3-element Array{Int64,1}:
 3
 4
 5
```

`Base.Iterators.cycle` — Function

```
cycle(iter)
```

An iterator that cycles through `iter` forever. If `iter` is empty, so is `cycle(iter)`.

Examples

```
julia> for (i, v) in enumerate(Iterators.cycle("hello"))
           print(v)
           i > 10 && break
       end
hellohelloh
```

`Base.Iterators.repeated` — Function

```
repeated(x[, n::Int])
```

An iterator that generates the value `x` forever. If `n` is specified, generates `x` that many times (equivalent to `take(repeated(x), n)`).

Examples

```
julia> a = Iterators.repeated([1 2], 4);

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

`Base.Iterators.product` — Function

```
product(iters...)
```

Return an iterator over the product of several iterators. Each generated element is a tuple whose `i`th element comes from the `i`th argument iterator. The first iterator changes the fastest.

Examples

```
julia> collect(Iterators.product(1:2, 3:5))
2×3 Array{Tuple{Int64,Int64},2}:
 (1, 3)  (1, 4)  (1, 5)
 (2, 3)  (2, 4)  (2, 5)
```

`Base.Iterators.flatten` — Function

```
flatten(iter)
```

Given an iterator that yields iterators, return an iterator that yields the elements of those iterators. Put differently, the elements of the argument iterator are concatenated.

Examples

```
julia> collect(Iterators.flatten((1:2, 8:9)))
4-element Array{Int64,1}:
 1
 2
 8
 9
```

`Base.Iterators.partition` — Function

```
partition(collection, n)
```

Iterate over a collection `n` elements at a time.

Examples

```
julia> collect(Iterators.partition([1,2,3,4,5], 2))
3-element Array{SubArray{Int64,1,Array{Int64,1},Tuple{UnitRange{Int64}},true},1
 [1, 2]
 [3, 4]
 [5]
```

`Base.Iterators.filter` — Function

```
Iterators.filter(flt, itr)
```

Given a predicate function `flt` and an iterable object `itr`, return an iterable object which upon iteration yields the elements `x` of `itr` that satisfy `flt(x)`. The order of the original iterator is preserved.

This function is *lazy*; that is, it is guaranteed to return in $\Theta(1)$ time and use $\Theta(1)$ additional space, and `flt` will not be called by an invocation of `filter`. Calls to `flt` will be made when iterating over the returned iterable object. These calls are not cached and repeated calls will be made when reiterating.

See `Base.filter` for an eager implementation of filtering for arrays.

Examples

```
julia> f = Iterators.filter(isodd, [1, 2, 3, 4, 5])
Base.Iterators.Filter{typeof(isodd),Array{Int64,1}}(isodd, [1, 2, 3, 4, 5])

julia> foreach(println, f)
1
3
5
```

`Base.Iterators.accumulate` — Function

```
Iterators.accumulate(f, itr; [init])
```

Given a 2-argument function `f` and an iterator `itr`, return a new iterator that successively applies `f` to the previous value and the next element of `itr`.

This is effectively a lazy version of `Base.accumulate`.

Julia 1.5

Keyword argument `init` is added in Julia 1.5.

Examples

```
julia> f = Iterators.accumulate(+, [1,2,3,4]);

julia> foreach(println, f)
1
3
6
10

julia> f = Iterators.accumulate(+, [1,2,3]; init = 100);

julia> foreach(println, f)
101
103
106
```

`Base.Iterators.reverse` — Function

```
Iterators.reverse(itr)
```

Given an iterator `itr`, then `reverse(itr)` is an iterator over the same collection but in the reverse order.

This iterator is "lazy" in that it does not make a copy of the collection in order to reverse it; see `Base.reverse` for an eager implementation.

Not all iterator types `T` support reverse-order iteration. If `T` doesn't, then iterating over `Iterators.reverse(itr::T)` will throw a `MethodError` because of the missing `iterate` methods for `Iterators.Reverse{T}`. (To implement these methods, the original iterator `itr::T` can be obtained from `r = Iterators.reverse(itr)` by `r.itr`.)

Examples

```
julia> foreach(println, Iterators.reverse(1:5))
5
4
3
2
1
```

`Base.Iterators.only` — Function

```
only(x)
```

Returns the one and only element of collection `x`, and throws an `ArgumentError` if the collection has zero or multiple elements.

See also: `first`, `last`.

Julia 1.4

This method requires at least Julia 1.4.

`Base.Iterators.peel` — Function

```
peel(iter)
```

Returns the first element and an iterator over the remaining elements.

Examples

```
julia> (a, rest) = Iterators.peel("abc");
julia> a
'a': ASCII/Unicode U+0061 (category Ll: Letter, lowercase)

julia> collect(rest)
2-element Array{Char,1}:
 'b': ASCII/Unicode U+0062 (category Ll: Letter, lowercase)
 'c': ASCII/Unicode U+0063 (category Ll: Letter, lowercase)
```

« Sorting and Related Functions C Interface »