Package ‘iterors’

May 18, 2023

Type  Package
Title  Fast, Compact Iterators and Tools
Version  1.0
Maintainer  Peter Meilstrup <peter.meilstrup@gmail.com>
Description  A fresh take on iterators in R. Designed to be cross-compatible with the 'iterators' package, but using the 'nextOr' method will offer better performance as well as more compact code. With batteries included: includes a collection of iterator constructors and combiners ported and refined from the 'iterators', 'itertools', and 'itertools2' packages.
Depends  R (>= 3.6)
Imports  rlang
Encoding  UTF-8
Suggests  reticulate, combinat, testthat (>= 3.0.0), foreach, iterators (>= 1.0.7), rmarkdown, knitr
URL  https://github.com/crowding/iterors,
     https://crowding.github.io/iterors/
License  GPL (>= 3)
RoxygenNote  7.2.3
VignetteBuilder  knitr
Config/testthat/edition  3
Config/testthat/parallel  false
NeedsCompilation  no
Author  Peter Meilstrup [cre, aut, cph],
        Folashade Daniel [aut],
        Revolution Analytics [aut, cph],
        Steve Weston [aut, cph],
        John A. Ramey [aut, cph],
        Kayla Schaefer [aut],
        Hadley Wickham [aut]
Repository  CRAN
Date/Publication  2023-05-18 08:40:02 UTC
R topics documented:

as.list.iteror ........................................... 3
concat .................................................. 4
consume .................................................. 5
count .................................................... 6
dotproduct .............................................. 7
hasNext .................................................. 7
icombinations .......................................... 8
icount ................................................... 10
idedup .................................................... 11
idiv ....................................................... 12
grid ........................................................ 13
ipermutations .......................................... 14
iread.table .............................................. 15
ireadBin .................................................. 16
ireaddf ................................................... 17
ireadLines ............................................... 18
iRNGStream .............................................. 19
irnorm ................................................... 20
is.iteror .................................................. 24
iseq ..................................................... 24
isplit ..................................................... 26
itabulate ............................................... 27
iteror .................................................... 27
iteror.array ............................................ 29
iteror.function ......................................... 31
i_apply .................................................. 32
i_break .................................................. 33
i_chunk .................................................. 34
i_concat ................................................ 35
i_dropwhile ............................................ 36
i_enumerate ............................................ 37
i_keep .................................................... 39
i_keepwhile ............................................ 40
i_limit .................................................... 41
i_map ..................................................... 42
i_mask .................................................... 43
i_pad ..................................................... 44
i_recycle ............................................... 45
i_rep ...................................................... 46
i_repeat ................................................ 47
i_rle ..................................................... 48
i_roundrobin .......................................... 49
i_slice ................................................... 49
i_starmap ............................................... 50
i_tee ..................................................... 51
i_timeout ................................................ 52
Collect all (or some given number of) values from an iteror, returning a vector of the given type.

### Description
Collect all (or some given number of) values from an iteror, returning a vector of the given type.

### Usage
```r
## S3 method for class 'iteror'
as.list(x, n = as.integer(2^31 - 1), ...)

## S3 method for class 'iteror'
as.double(x, n = as.integer(2^31 - 1), ...)

## S3 method for class 'iteror'
as.numeric(x, n = as.integer(2^31 - 1), ...)

## S3 method for class 'iteror'
as.logical(x, n = as.integer(2^31 - 1), ...)

## S3 method for class 'iteror'
as.character(x, n = as.integer(2^31 - 1), ...)

## S3 method for class 'iteror'
as.vector(x, mode)
```

### Arguments
- `x`: an iterable object
- `n`: the maximum number of elements to return.
- `...`: Unused arguments will throw an error.
- `mode`: What mode to use for the output vector.
Value

The returned value will be n elements long if the iterator did not stop.

See Also

concat take

---

**concat**

Concatenate contents of multiple iterators into a vector.

### Description

concat collects all values from an iterable object, and pastes them end to end into one vector. In other words, `concat` is to `as.list.iteror` as `c` is to `list`.

### Usage

```r
concat(obj, mode = "list", n = Inf, ...) 
## Default S3 method: 
concat(obj, mode = "list", n = as.integer(2^31 - 1), ...) 
## S3 method for class 'iteror'
concat(obj, mode = "list", n = Inf, length.out = Inf, ...)
```

### Arguments

- `obj` An iteror.
- `mode` The mode of vector to return.
- `n` The maximum number of times to call `nextOr(obj)`.
- `...` passed along to `iteror` constructor.
- `length.out` The target size of the output vector (after results have been pasted together). If the iterator ends (or emits n results) before emitting this many elements, the result will be shorter than `length.out`. If the iterator does not end early, the output will have at least `length.out` elements, and possibly more, as the entire last chunk will be included.

### Value

a vector with mode `mode`.

### Examples

```r
it <- i_apply(icount(), seq_len)  # [1], [1, 2], [1, 2, 3], ...
concat(it, n=4, mode="numeric")  # [1, 1, 2, 1, 2, 3, 1, 2, 3, 4]
concat(it, length.out=4, mode="numeric")  # [1, 1, 2, 1, 2, 3, 1, 2, 3, 4]
```
consume

Consumes the first n elements of an iterator

Description
Advances the iterator n-steps ahead without returning anything.

Usage
consume(obj, n = Inf, ...)

## S3 method for class 'iteror'
consume(obj, n = Inf, ...)

Arguments
obj an iterable object
n The number of elements to consume.
... passed along to iteror constructor.

Value
obj, invisibly.

See Also
take collect

Examples
it <- iteror(1:10)
# Skips the first 5 elements
consume(it, n=5)
# Returns 6
nextOr(it, NA)

it2 <- iteror(letters)
# Skips the first 4 elements
consume(it2, 4)
# Returns 'e'
nextOr(it2, NA)
count

Consumes an iterator and computes its length

Description

Counts the number of elements in an iterator. NOTE: The iterator is consumed in the process.

Usage

count(object, ...)

Arguments

object an iterable object

... passed along to iteror constructor.

Value

the number of elements in the iterator

See Also

take consume as.list.iteror

Examples

count(1:5) == length(1:5)

it <- iteror(1:5)
count(it) == length(1:5)

it2 <- i_chain(1:3, 4:5, 6)
count(it2)

it3 <- i_chain(1:3, levels(iris$Species))
count(it3)
**dotproduct**

*Computes the dot product of two iterable objects.*

**Description**

Returns the dot product of two numeric iterables of equal length.

**Usage**

```r
dotproduct(vec1, vec2)
```

**Arguments**

- `vec1` the first
- `vec2` the second iterable object

**Value**

the dot product of the iterators

**Examples**

```r
it <- iteror(1:3)
it2 <- iteror(4:6)
dotproduct(it, it2) # 32

it <- iteror(1:4)
it2 <- iteror(7:10)
dotproduct(1:4, 7:10) # 90
```

**hasNext**

*Does This Iterator Have A Next Element*

**Description**

`wrapped <- ihasnext(obj)` wraps an `iteror` object with the `ihasNext` class. Then `hasNext(wrapped)` will indicate if the iterator has another element.

**Usage**

```r
hasNext(obj, ...)

ihasNext(obj, ...)
```
icombinations

Arguments

- obj: an iterable
- ... extra arguments may be passed along to iteror.

Details

A class ihasNext was introduced in the itertools package to try to reduce the boilerplate around extracting the next value using iterators::nextElem. ihasNext is included in iterors for backward compatibility with iterator code; however, it is less needed when using the nextOr iteration method, as you can directly give an action to take at end of iteration.

Value

Logical value indicating whether the iterator has a next element.

Examples

# The bad old style of consuming an iterator in a loop with `nextElem`:
it <- ihasNext(iteror(c('a', 'b', 'c')))
tryCatch(repeat {
  print(iterators::nextElem(it))
}, error=function(err) {
  if (conditionMessage(err) != "StopIteration")
    stop(err)
})

# with ihasNext, this became:
it <- ihasNext(iteror(c('a', 'b', 'c')))
while (hasNext(it))
  print(iterators::nextElem(it))

# But using `nextOr` all you need is:
iteror(c('a', 'b', 'c'))
repeat print(nextOr(it, break))

icombinations

Iterator that generates all combinations of a vector taken m at a time.

Description

Constructs an iterator generates all combinations of a vector taken m at a time. This function is similar to combn.

Usage

icombinations(object, m, replacement = FALSE)
**icombinations**

**Arguments**

- object: vector
- m: the length of each combination
- replacement: Generate combinations with replacement? Default: no.

**Details**

By default, the combinations are **without replacement** so that elements are not repeated. To generate combinations **with replacement**, set `replacement=TRUE`.

The function implementation is loosely based on the `combinations` function from Python’s `itertools`. Combinations with replacement are based on `combinations_with_replacement` from the same Python library.

**Value**

iterator that generates all combinations of `object`

**Examples**

```r
eight <- icombinations(1:4, m=2)

eight <- icombinations(1:4, m=2, replacement=TRUE)

it3 <- icombinations(1:5, m=2)

as.list(it3)

utils::combn(x=1:5, m=2, simplify=FALSE)
```
icount  

Counting Iterators

Description

Returns an iterator that counts starting from one.

icountn(vn) takes a vector specifying an array size, and returns an iterator over array indices. Each returned element is a vector the same length as vn, with the first index varying fastest. If vn has a names attribute the output will have the same names.

Usage

icount(count = Inf, ..., recycle = FALSE, chunkSize, chunks)

icountn(vn, ..., recycle = FALSE, chunkSize, chunks, rowMajor = TRUE)

Arguments

count  

number of times that the iterator will fire. Use NA or Inf to make an iterator that counts forever.

...  

Undocumented

recycle  

Whether to restart the count after finishing.

chunkSize  

How many values to return from each call to nextOr().

chunks  

How many chunks to split the input. Either chunks or chunkSize may be given but not both.

vn  

A vector of integers.

rowMajor  

If TRUE (default), the earliest indices will cycle fastest; if FALSE, last indices cycle fastest.

Details

Originally from the iterators package.

Value

The counting iterator.

See Also

For more control over starting number and step size, see iseq.
```r
# create an iterator that counts from 1 to 3.
it <- icount(3)
nextOr(it)
nextOr(it)
nextOr(it)
nextOr(it, NULL)  # expect NULL

x <- icount(5)
repeat print(nextOr(x, break))

it2 <- icount(100)
all.equal(as.numeric(it2), 1:100)
as.list(icountn(c(2, 3)))
```

```r
identdup

Drop duplicated items from an iterator.

Description

Constructs an iterator that removes runs of repeated elements from the underlying iterator. Order of the elements is maintained. Only the element just seen is remembered for determining whether to drop.

Usage

```r
identdup(object, cmp = identical, ...)
```

Arguments

- `object`: an iterable object
- `cmp`: A function to use for comparison.
- `...`: passed along to `iteror(object, ...)`

Details

Originated as `itertools2::iunique_lastseen` object.

Value

an iterator that skips over duplicate items from the underlying iterator.

See Also

`i_rle`
```
Examples

```r
it <- i_chain(rep(1, 4), rep(2, 5), 4:7, 2)
it_i_unique <- idedup(it)
as.list(it_i_unique) # 1 2 4 5 6 7 2

it2 <- iteror(c('a', 'a', 'A', 'a', 'a', 'V'))
i_dedupe <- idedup(it2)
as.list(idedup) # a A a V
```

---

**idiv**  
**Dividing Iterator**

Description

Returns an iterator dividing a value into integer chunks, such that \( \text{sum(idiv}(n, \ldots) = \text{floor}(n) \)

Usage

\[
idiv(\text{count}, \ldots, \text{recycle} = \text{FALSE}, \text{chunkSize}, \text{chunks})
\]

Arguments

- **count**: The total
- **...**: Unused.
- **recycle**: Whether to restart the count after finishing.
- **chunkSize**: the maximum size of the pieces that \( n \) should be divided into. This is useful when you know the size of the pieces that you want. If specified, then \( \text{chunks} \) should not be.
- **chunks**: the number of pieces that \( n \) should be divided into. This is useful when you know the number of pieces that you want. If specified, then \( \text{chunkSize} \) should not be.

Details

Originally from the iterators package.

Value

The dividing iterator.
Examples

    # divide the value 10 into 3 pieces
    it <- idiv(10, chunks = 3)
    nextOr(it)
    nextOr(it)
    nextOr(it)
    nextOr(it, NULL)  # expect NULL

    # divide the value 10 into pieces no larger than 3
    it <- idiv(10, chunkSize = 3)
    nextOr(it)
    nextOr(it)
    nextOr(it)
    nextOr(it)
    nextOr(it, NULL)  # end of iterator

---

**igrid**  
*Iterator that covers the Cartesian product of the arguments.*

Description

Given a number of vectors as arguments, constructs an iterator that enumerates the Cartesian product of all arguments.

Usage

    igrid(
        ...,  
        recycle = FALSE,  
        chunkSize,  
        chunks,  
        simplify = FALSE,  
        rowMajor = TRUE  
    )

Arguments

    ...  
    recycle  
    chunkSize  
    chunks  
    simplify  
    rowMajor

    Named vectors to iterate over.
    If TRUE, the iterator starts over on reaching the end.
    Optional; how many rows to return in each step.
    Optional; how many chunks to divide the input into.
    If TRUE, inputs are coerced to a common data type and results are returned in a vector (or matrix if chunking is enabled). If FALSE, results are returned as a list (or data.frame if chunking).
    If TRUE, the left-most indices change fastest. If FALSE the rightmost indices change fastest.
Details

Although they share the same end goal, igrid can yield drastic memory savings compared to expand.grid.

Value

an iterator that iterates through each element from the Cartesian product of its arguments.

Examples

# Simulate a doubly-nested loop with a single while loop
it <- igrid(a=1:3, b=1:2)
repeat {
  x <- nextOr(it, break)
  cat(sprintf('a = %d, b = %d
', x$a, x$b))
}

it <- igrid(x=1:3, y=4:5)
nextOr(it, NA) # list(x=1, y=4)
nextOr(it, NA) # list(x=1, y=5)
nextOr(it, NA) # list(x=2, y=4)
nextOr(it, NA) # list(x=2, y=5)
nextOr(it, NA) # list(x=3, y=4)
nextOr(it, NA) # list(x=3, y=5)

# Second Cartesian product
nextOr(it, NA) # list(x=1, y=4)
nextOr(it, NA) # list(x=1, y=5)
nextOr(it, NA) # list(x=2, y=4)
nextOr(it, NA) # list(x=2, y=5)
nextOr(it, NA) # list(x=3, y=4)
nextOr(it, NA) # list(x=3, y=5)

# igrid is an iterator equivalent to base::expand.grid()
# Large data.frames are not created unless the iterator is manually consumed
a <- 1:2
b <- 3:4
c <- 5:6
it3 <- igrid(a=a, b=b, c=c)
df_igrid <- do.call(rbind, as.list(it3))
df_igrid <- data.frame(df_igrid)

# Compare df_igrid with the results from base::expand.grid()
base::expand.grid(a=a, b=b, c=c)
Description

Constructs an iterator that generates all permutations of an iterable object. By default, full-length permutations are generated. If \( m \) is specified, successive \( m \) length permutations are instead generated.

Usage

\[
\text{ipermutations}(\text{object}, m = \text{NULL})
\]

Arguments

- \text{object} vector
- \text{m} length of permutations. By default, full-length permutations are generated.

Details

The implementation is loosely based on that of Python’s itertools.

Value

Iterator that generates all permutations of \text{object}

Examples

\[
\text{it} \leftarrow \text{ipermutations}(1:3)
\]

\[
\text{nextOr(it, NA)} # c(1, 2, 3)
\]

\[
\text{nextOr(it, NA)} # c(1, 3, 2)
\]

\[
\text{nextOr(it, NA)} # c(3, 1, 2)
\]

\[
\text{nextOr(it, NA)} # c(3, 2, 1)
\]

\[
\text{nextOr(it, NA)} # c(2, 3, 1)
\]

\[
\text{nextOr(it, NA)} # c(2, 1, 3)
\]

\[
\text{it2} \leftarrow \text{ipermutations(letters[1:4])}
\]

\[
\text{# 24 = 4! permutations of the letters a, b, c, and d}
\]

\[
\text{as.list(it2)}
\]

---

\text{iread.table} \\
\text{Iterator over Rows of a Data Frame Stored in a File}

Description

Returns an iterator over the rows of a data frame stored in a file in table format. It is a wrapper around the standard \text{read.table} function.

Usage

\[
\text{iread.table(file, ..., verbose = FALSE)}
\]
ireadBin

Create an iterator to read binary data from a connection

Description

Create an iterator to read binary data from a connection.

Arguments

- **file**
  - the name of the file to read the data from.
- **...**
  - all additional arguments are passed on to the read.table function. See the documentation for read.table for more information.
- **verbose**
  - logical value indicating whether or not to print the calls to read.table.

Details

Originally from the iterators package.

Value

The file reading iterator.

Note

In this version of iread.table, both the read.table arguments header and row.names must be specified. This is because the default values of these arguments depend on the contents of the beginning of the file. In order to make the subsequent calls to read.table work consistently, the user must specify those arguments explicitly. A future version of iread.table may remove this requirement.

See Also

- read.table

ireadBin(  
  con,  
  what = "raw",  
  n = 1L,  
  size = NA_integer_,  
  signed = TRUE,  
  endian = .Platform$endian,  
  ipos = NULL  
)
ireaddf

Arguments

con
A connection object or a character string naming a file or a raw vector.

what
Either an object whose mode will give the mode of the vector to be read, or a character vector of length one describing the mode: one of “numeric”, “double”, “integer”, “int”, “logical”, “complex”, “character”, “raw”. Unlike readBin, the default value is “raw”.

n
integer. The (maximal) number of records to be read each time the iterator is called.

size
integer. The number of bytes per element in the byte stream. The default, ‘NA_integer_’, uses the natural size.

signed
logical. Only used for integers of sizes 1 and 2, when it determines if the quantity on file should be regarded as a signed or unsigned integer.

endian
The endian-ness (“big” or “little”) of the target system for the file. Using “swap” will force swapping endian-ness.

ipos
iterable. If not NULL, values from this iterable will be used to do a seek on the file before calling readBin.

Details

Originally from the itertools package.

Value

An iterator reading binary chunks from the connection.

Examples

zz <- file("testbin", "wb")
writeBin(1:100, zz)
close(zz)

it <- ihasNext(ireadBin("testbin", integer(), 10))
repeat print(nextOr(it, break))
unlink("testbin")

ireaddf

Create an iterator to read data frames from files

Description

Create an iterator to read data frames from files.

Usage

ireaddf(filenames, n, start = 1, col.names, chunkSize = 1000)
ireadLines

Arguments

filenames   Names of files contains column data.
n         Maximum number of elements to read from each column file.
start       Element to start reading from.
col.names   Names of the columns.
chunkSize   Number of rows to read at a time.

Details

Originally from the itertools package.

Value

An iterator yielding data.frame objects with up to n rows.

Description

Returns an iterator over the lines of text from a connection. It is a wrapper around the standard readLines function.

Usage

ireadLines(con, n = 1, ...)

Arguments

con       a connection object or a character string.
n         integer. The maximum number of lines to read. Negative values indicate that one should read up to the end of the connection. The default value is 1.
...       passed on to the readLines function.

Details

Originally from the iterators package.

Value

The line reading iterator.

See Also

readLines
Examples

```r
# create an iterator over the lines of COPYING
it <- ireadLines(file.path(R.home(), "COPYING"))
nextOr(it)
nextOr(it)
nextOr(it)
```

Description

The `iRNGStream` creates a sequence of random number seeds that are very "far apart" (2^127 steps) in the overall random number sequence, so that each can be used to make a parallel, pseudo-independent random iterator. This uses `parallel::nextRNGStream` and the "L'Ecuyer-CMRG" generator.

Usage

`iRNGStream(seed)`

Arguments

- `seed` Either a single number to be passed to `set.seed` or a

Details

`iRNGSubStream` creates seeds that are somewhat less far apart (2^76 steps), which might be used as "substream" seeds.

Originally from the `itertools` package.

Value

An iterator which produces seed values. vector to be passed to `nextRNGStream` or `nextRNGSubStream`. An iterator which yields successive seed values.

References

For more details on the L'Ecuyer-CMRG generator, see `vignette("parallel", package="parallel")`.

See Also

`set.seed`, `nextRNGStream`, `nextRNGSubStream`
Examples

```r
global.seed <- .Random.seed

rng.seeds <- iRNGStream(313)
print(nextOr(rng.seeds))
print(nextOr(rng.seeds))

# create three pseudo-independent and
# reproducible random number streams
it1 <- isample(c(0, 1), 1, seed=nextOr(rng.seeds))
it2 <- isample(c(0, 1), 1, seed=nextOr(rng.seeds))
it3 <- isample(c(0, 1), 1, seed=nextOr(rng.seeds))

all(.Random.seed == global.seed)

all(.Random.seed == global.seed)

take(it1, 5, "numeric") # 0 0 0 1 1

take(it2, 5, "numeric") # 0 1 1 1 1

take(it3, 5, "numeric") # 1 1 1 0 0

# none of this affects the global seed
all(global.seed == .Random.seed)

# Compute random numbers in three parallel processes with three
# well-separated seeds. Requires package "foreach"
library(foreach)

foreach(1:3, rseed=iRNGSubStream(1970), .combine='c') %dopar% {
  RNGkind("L'Ecuyer-CMRG") # would be better to initialize workers only once
  assign(".Random.seed", rseed, pos=.GlobalEnv)
  runif(1)
}
```

---

`irnorm`  
*Random Number Iterators*

**Description**

These functions each construct an iterator that produces random numbers of various distributions. Each one is a wrapper around a base R function.

**Usage**

```r
irnorm(
  n,
  mean = 0,
  sd = 1,
  count = Inf,
  independent = !missing(seed) || !missing(kind),
)```
irnorm

seed = NULL,
kind = NULL,
normal.kind = NULL,
sample.kind = NULL
)

irbinom(
  n,
  size,
  prob,
  count = Inf,
  independent = !missing(seed) || !missing(kind),
  seed = NULL,
  kind = NULL,
  normal.kind = NULL,
  sample.kind = NULL
)

irnbinom(
  n,
  size,
  prob,
  mu,
  count = Inf,
  independent = !missing(seed) || !missing(kind),
  seed = NULL,
  kind = NULL,
  normal.kind = NULL,
  sample.kind = NULL
)

irpois(
  n,
  lambda,
  count = Inf,
  independent = !missing(seed) || !missing(kind),
  seed = NULL,
  kind = NULL,
  normal.kind = NULL,
  sample.kind = NULL
)

isample(
  x,
  size,
  replace = FALSE,
  prob = NULL,
  count = Inf,
independent = !missing(seed) || !missing(kind),
seed = NULL,
kind = NULL,
normal.kind = NULL,
sample.kind = NULL
)

irunif(
  n,
  min = 0,
  max = 1,
  count = Inf,
  independent = !missing(seed) || !missing(kind),
  seed = NULL,
  kind = NULL,
  normal.kind = NULL,
  sample.kind = NULL
)

Arguments

n How many samples to compute per call; see e.g. rnorm.
mean see rnorm.
sd see rnorm.
count number of times that the iterator will fire. If not specified, it will fire values forever.
independent If TRUE, this iterator will keep its own private random state, so that its output is reproducible and independent of anything else in the program; this comes at some performance cost. Default is FALSE unless seed or kind are given. If independent=TRUE but neither seed nor kind are specified, we will use the "L’Ecuyer-CMRG" generator with a seed value taken from a package-private instance of iRNGStream.
seed A specific seed value for reproducibility. If given, independent=TRUE is implied. This can be a single number (which will be passed to set.seed(seed, kind, normal.kind, sample.kind); it can also be a vector containing a complete, valid state for .Random.seed. If the latter, arguments kind, etc. are ignored.
kind Which random number algorithm to use; passed along to set.seed. If given, independent=TRUE is implied.
normal.kind Passed along to set.seed.
sample.kind Passed along to set.seed.
size see e.g. rbinom.
prob see e.g. rbinom.
mu see rbinom.
lambda see rpois.
**irnorm**

- `x` see `isample`
- `replace` see `isample`
- `min` see `runif`
- `max` see `runif`

**Details**

Originally from the *iterators* package.

**Value**

An iterator that is a wrapper around the corresponding random number generator function.

**See Also**

If you are creating multiple independent iterators, *iRNGStream* will create well-separated seed values, which may help avoid spurious correlations between iterators.

**Examples**

```r
# create an iterator that returns three random numbers
it <- irnorm(1, count = 3)
nextOr(it)
nextOr(it)
nextOr(it)
nextOr(it, NULL)

# iterators created with a specific seed will make reproducible values
it <- irunif(n=1, seed=314, kind="L'Ecuyer-CMRG")
nextOr(it) # 0.4936700
nextOr(it) # 0.5103891
nextOr(it) # 0.2338745

# the iRNGStream produces a sequence of well separated seed values,
rng.seeds <- iRNGStream(313)
it1 <- isample(c(0, 1), 1, seed=nextOr(rng.seeds))
it2 <- isample(c(0, 1), 1, seed=nextOr(rng.seeds))
it3 <- isample(c(0, 1), 1, seed=nextOr(rng.seeds))
take(it1, 5, "numeric") # 0 1 0 0 1
take(it2, 5, "numeric") # 0 1 0 0 0
take(it3, 5, "numeric") # 0 0 0 1 1
```
is.iteror

is.iteror indicates if an object is an iteror.

Description

is.iteror indicates if an object is an iteror.

Usage

is.iteror(x)

Arguments

x any object.

Value

TRUE if the object has class iteror.

Examples

it <- iteror(1:3)
stopifnot(is.iteror(it))
repeat {
  print(nextOr(it, break))
  ...
}

iseq

Iterators for sequence generation

Description

Constructs iterators that generate regular sequences that follow the seq family.

Usage

iseq()
  from = 1,
  to = NULL,
  by = NULL,
  length_out = NULL,
  along_with = NULL,
  ...
  recycle = FALSE,
iseq

chunkSize, chunks

iseq_along(along_with, ...)

Arguments

* from the starting value of the sequence.
* to the end value of the sequence.
* by increment of the sequence.
* length_out desired length of the sequence. A non-negative number, which for seq will be rounded up if fractional.
* along_with the length of the sequence will match the length of this ...
* recycle Whether to restart the sequence after it reaches to.
* chunkSize Optional; return this many values per call.
* chunks Optional; return this many chunks.

Details

The iseq function generates a sequence of values beginning with from and ending with to. The sequence of values between are determined by the by, length_out, and along_with arguments. The by argument determines the step size of the sequence, whereas length_out and along_with determine the length of the sequence. If by is not given, then it is determined by either length_out or along_with. By default, neither are given, in which case by is set to 1 or -1, depending on whether to > from.

Value

an iteror.

See Also

icount icountn

Examples

it <- iseq(from=2, to=5)
unlist(as.list(it)) == 2:5
isplit  Split Iterator

Description

Returns an iterator that divides the data in the vector \( x \) into the groups defined by \( f \).

Usage

\[
isplit(x, f, \text{drop} = \FALSE, \ldots)
\]

Arguments

- \( x \): vector or data frame of values to be split into groups.
- \( f \): a factor or list of factors used to categorize \( x \).
- \( \text{drop} \): logical indicating if levels that do not occur should be dropped.
- \( \ldots \): current ignored.

Details

Originally from the iterators package.

Value

The split iterator.

See Also

split

Examples

```r
x <- rnorm(200)
f <- factor(sample(1:10, length(x), replace = TRUE))

it <- isplit(x, f)
expected <- split(x, f)

for (i in expected) {
  actual <- nextOr(it, break)
  stopifnot(actual$value == i)
}
```
**Description**

Constructs an iterator that maps a given function over an indefinite sequence of numeric values. The input the function \( f \) is expected to accept a single numeric argument. The sequence of arguments passed to \( f \) begin with \( \text{start} \) and are incremented by \( \text{step} \).

**Usage**

\[
\text{itabulate}(f, \ \text{start} = 1, \ \text{step} = 1)
\]

**Arguments**

- **f** the function to apply
- start sequence's initial value
- step sequence's step size

**Value**

an iterator that returns the mapped values from the sequence

**Examples**

\[
\text{it} \leftarrow \text{itabulate}(f=\text{function}(x) \ x + 1) \\
\text{take}(\text{it}, 4) \# 2 \ 3 \ 4 \ 5
\]

\[
\text{it2} \leftarrow \text{itabulate}(f=\text{function}(x) \ x^2, \ \text{start}=-3) \\
\text{take}(\text{it2}, 6) \# 9 \ 4 \ 1 \ 0 \ 1 \ 4
\]

\[
\text{it3} \leftarrow \text{itabulate}(\text{abs}, \ \text{start}=-5, \ \text{step}=2) \\
\text{take}(\text{it3}, 6) \# 5 \ 3 \ 1 \ 1 \ 3 \ 5
\]

\[
\text{it4} \leftarrow \text{itabulate}(\exp, \ \text{start}=6, \ \text{step}=-2) \\
\text{take}(\text{it4}, 4) \# \exp(c(6, \ 4, \ 2, \ 0))
\]

---

**iteror**

*Make an iteror from a given object.*

**Description**

\( \text{it} \leftarrow \text{iteror}(\text{obj}, \ldots) \) is a generic constructor that creates objects of class "iteror" from its input. An iteror outputs a single element of a sequence each time you call \( \text{nextOr}(\text{it}) \). Different iteror methods exist for different data types and may take different optional arguments as listed in this page.
Usage

iteror(obj, ...)

## S3 method for class 'iter'
iteror(obj, ...)

## Default S3 method:
iteror(obj, ..., recycle = FALSE, chunkSize, chunks)

## S3 method for class 'connection'
iteror(obj, ...)

Arguments

obj An object to iterate with.
...
... Different iteror methods may take additional options depending on the class
of obj.
recycle a boolean describing whether the iterator should reset after running through all
its values.
chunkSize How many elements (or slices) to include in each chunk.
chunks Split the input into this many chunks.

Details

When called, an iteror may either return a new value or stop. The way an iteror signals a stop is that
it does whatever you write in the argument or. For instance you can write or = break to exit a loop.
Summing over an iteror this way looks like:

```r
sum <- 0
it <- iteror(iseq(0, 100, 7))
repeat {
  sum <- sum + nextOr(it, break)
}
```

Another way to use the "or" argument is to give it a sentinel value; that is, a special value that you
will interpret as end of iteration. If the result of calling nextOr is identical() to the value you
provided, then you know the iterator has ended. This pattern looks like:

```r
sum <- 0
stopped <- new.env()
it <- iteror(iseq(0, 100, 7))
repeat {
  val <- nextOr(it, stopped)
  if (identical(val, stopped)) break
  sum <- sum + val
}
(Here I’m using new.env() as a sentinel value. In R it is commonplace to use NULL or NA as a kind of sentinel value, but that only works until you have an iterator that needs to yield NULL itself. A safer alternative is to use a local, one-shot sentinel value; new.env() is ideal, as it constructs an object that is not identical to any other object in the R session.)

Note that iteror objects are simply functions with a class attribute attached, and all nextOr.iteror does is call the function. So if you were in the mood, you could skip calling nextOr involving S3 dispatch and instead call the iteror directly. If you take this approach, make sure you have called iteror() first to ensure that you have a true iteror object.

```r
sum <- 0
it <- iteror(iseq(0, 100, 7))
repeat sum <- sum + it(or=break)
sum
#> [1] 735
```

To create iterors with custom-defined behavior, see iteror.function.

**Value**

an object of classes ‘iteror’ and ‘iter’.

The method iteror.iter wraps an iterators::iter object and returns an iteror.

The default method iteror.default treats obj as a vector to yield values from.

**See Also**

iteror.array iteror.function iteror.data.frame

---

### iteror.array

**Iterate over an array or data frame by a specified dimension.**

**Description**

Iterate over an array or data frame by a specified dimension.

**Usage**

```r
## S3 method for class 'array'
iteror(
  obj,
  ..., 
  by = c("cell", "row", "column"),
  chunkSize,
  chunks,
  recycle = FALSE,
  drop = FALSE,
  rowMajor = TRUE
)```

## S3 method for class 'matrix'
iteror(
  obj,
  ..., 
  by = c("cell", "row", "column"),
  chunkSize,
  chunks,
  recycle = FALSE,
  drop = FALSE,
  rowMajor = TRUE
)

## S3 method for class 'data.frame'
iteror(obj, ..., recycle = FALSE, chunkSize, chunks, by = c("column", "row"))

**Arguments**

- **obj**
  An object to iterate over.

- **...**
  Undocumented.

- **by**
  Which dimension to slice an array or data frame by. Can be "cell", "row", "column", or numeric dimensions.

- **chunkSize**
  The thickness of the slice to take along the specified dimension.

- **chunks**
  How many slices to take.

- **recycle**
  If TRUE, the iteror starts over on reaching the end.

- **drop**
  Whether to drop the array dimensions enumerated over.

- **rowMajor**
  If TRUE, will return slices in order with the first indices varying fastest (same as in `i enumerate`).

**Value**

an iteror yielding from obj along the specified dimensions.

**Examples**

```r
l <- iteror(letters, chunkSize=7)
as.list(l)

a <- array(1:8, c(2, 2, 2))

# iterate over all the slices
it <- iteror(a, by=3)
as.list(it)

# iterate over all the columns of each slice
it <- iteror(a, by=c(2, 3))
as.list(it)
```
# iterate over all the rows of each slice
it <- iteror(a, by=c(1, 3))
as.list(it)

---

**iteror.function**

*Construct an iteror object with custom-programmed behavior.*

**Description**

Pass `obj` a function that has a first argument named "or". In writing this function, you can maintain state by using enclosed variables and update using `<<-`. Whatever value `obj()` returns is the next element of the iteror. Treat argument `or` as a lazy value; do not touch it until until you need to signal end of iteration; to signal end of iteration, force and immediately return `or`.

**Usage**

```r
## S3 method for class `function`
iteror(obj, ..., catch, sentinel, count)
```

**Arguments**

- `obj` A function. It should have having an argument named "or"
- `...` Undocumented.
- `catch` If `obj` does not have an `or` argument, specify e.g. `catch="StopIteration"` to interpret that an error with that message as end of iteration.
- `sentinel` If `obj` does not have an `or` argument, you can specify a special value to watch for end of iteration. Stop will be signaled if the function result is `identical()` to `sentinel`.
- `count` If `obj` does not have an `or` argument, you can specify how many calls before stop iteration, or give `NA` or `Inf` to never stop.

**Details**

You can also provide `obj` a simple function of no arguments, as long as you specify one of `catch`, `sentinel`, or `count` to specify how to detect end of iteration.

**Value**

An object of mode "function" and class "iteror".

An `iteror` which calls the given function to produce values.
Examples

# an iterator that counts from start to stop
irange <- function(from=1, to=Inf) {
  current <- from
  iteror(function(or) {
    if (current > to) {
      return(or)
    } else {
      tmp <- current
      current <<- current + 1
      tmp
    }
  })
}
it <- irange(5, 10)
as.vector(it, "numeric")

# an endless random number generator
irand <- function(min, max) {
  iteror(function() runif(1, min=min, max=max), count=Inf)
}
take(irand(5, 10), 10)

---

i_apply

Apply a function to each element of an iterator.

Description

i_apply(obj, f) returns the iteror that applies f to each element of the given iterable obj. It is an iterator equivalent of lapply.

Usage

i_apply(obj, f, ...)

Arguments

obj an iterable.
f a function
... Additional arguments will be passed along to f

Value

An iteror.
**i_break**

Create an iterator that can be told to stop.

**Description**

Create an iterator that iterates over another iterator until a specified function returns FALSE. This can be useful for breaking out of a foreach loop, for example.

**Usage**

```r
i_break(iterable, finished, ...)
```

**Arguments**

- `iterable` : Iterable to iterate over.
- `finished` : Function that returns a logical value. The iterator stops when this function returns FALSE.
- `...` : Further arguments forwarded to `iteror`.

**Details**

Originally from the `itertools` package.

**Value**

an iteror which will stop when `finished()` is TRUE

**Examples**

```r
# See how high we can count in a tenth of a second
mkfinished <- function(time) {
  starttime <- proc.time()[3]
  function() proc.time()[3] > starttime + time
}
length(as.list(i_break(icount(), mkfinished(0.1)))))
```
i_chunk

Combine an iterator’s values into chunks.

Description

Create an iterator that issues lists of values from the underlying iterable. This is useful for manually “chunking” values from an iterable.

Usage

i_chunk(iterable, size, mode = "list", fill, ...)

Arguments

iterable
Iterable to iterate over.

size
Maximum number of values from iterable to return in each value issued by the resulting iterator.

mode
Mode of the objects returned by the iterator.

fill
Value to use to pad the last chunk to size, if it is short. If missing, no padding will be done.

... Further arguments will be forwarded to iterator(iterable, ...).

Value

an iterator that yields items of length size and mode mode.

See Also

iterator.default

Argument size does not need to be an integer, for instance a chunk of 3.5 will produce chunks of sizes 3 and 4 alternating. The precise behavior will be subject to floating point precision.

Examples

# Split the vector 1:10 into “chunks” with a maximum length of three
it <- i_chunk(1:10, 3)
repeat print(unlist(nextOr(it, break)))

# Same as previous, but return integer vectors rather than lists
it <- i_chunk(1:10, 3, mode='integer')
repeat print(unlist(nextOr(it, break)))

it <- i_chunk(iterators::iter(1:5), 2, fill=NA)
# List: list(1, 2, 3)
nextOr(it, NULL)
# List: list(4, 5, NA)
```
nextOr(it, NULL)

it2 <- i_chunk(levels(iris$Species), 4, fill="weeee")
# Returns: list("setosa", "versicolor", "virginica", "weeee")
nextOr(it2, NA)
```

---

**i_concat**

*Iterator that chains multiple arguments together into a single iterator*

---

**Description**

`i_concat(obj)` takes an iterable that returns iterables, and chains together all inner values of iterables into one iterator. Analogous to `unlist(recursive=FALSE)`.

`i_chain` for iterators is analogous to `c()` on vectors. `i_chain` constructs an iterator that returns elements from the first argument until it is exhausted, then elements from the next argument, and so on until all arguments have been exhausted.

**Usage**

```r
i_concat(obj, ...)
i_chain(...)
```

**Arguments**

- `obj`: an iterable.
- `...`: multiple iterable arguments

**Value**

Iterator that iterates through each argument in sequence

**Author(s)**

Peter Meilstrup

**Examples**

```r
it <- i_chain(1:3, 4:5, 6)
as.list(it)

it2 <- i_chain(1:3, levels(iris$Species))
as.list(it2)
```
**i_dropwhile**

*Iterator that drops elements until the predicate function returns FALSE*

---

**Description**

Constructs an iterator that drops elements from the iterable object as long as the predicate function is true; afterwards, every element of iterable object is returned.

**Usage**

\[
i\_dropwhile(\text{object}, \text{predicate}, \ldots)\]

**Arguments**

- **object**
  - an iterable object
- **predicate**
  - a function that determines whether an element is TRUE or FALSE. The function is assumed to take only one argument.
- ... Further arguments forwarded to *iteror*.

**Details**

Because the iterator does not return any elements until the predicate first becomes false, there may have a lengthy start-up time before elements are returned.

**Value**

An *iteror* object.

**Examples**

```r
# Filters out numbers exceeding 3
not_too_large <- function(x) {
  x <= 3
}

it <- i_dropwhile(1:8, not_too_large)
as.list(it)

# Same approach but uses an anonymous function

it2 <- i_dropwhile(seq(2, 20, by=2), function(x) x <= 10)
as.list(it2)
```
**i_enumerate**

**Iterator that returns the elements of an object along with their indices**

**Description**

Constructs an iterator that returns the elements of an object along with each element’s indices. Enumeration is useful when looping through an object and a counter is required.

The `i_enumerate` method for arrays allows splitting an array by arbitrary margins, including by multiple margins. The index element returned will be a vector (or if chunking is used, a matrix) of indices.

**Usage**

```
i_enumerate(obj, ...)

i_enum(obj, ...)
```

## Default S3 method:
```
i_enumerate(obj, ..., recycle = FALSE, chunkSize, chunks)
```

```
i_enumerate(obj, ...)
```

## S3 method for class 'array'
```
i_enumerate(
  obj,
  ..., recycle = FALSE, chunkSize, chunks,
  by = c("cell", "row", "column"),
  rowMajor = TRUE,
  drop = FALSE
)
```

**Arguments**

- `obj`: object to return indefinitely.
- `...`: Undocumented.
- `recycle`: Whether to restart the iterator after finishing the array.
- `chunkSize`: How large a chunk to take along the specified dimension.
- `chunks`: How many chunks to divide the array into.
- `by`: Which array margins to iterate over. Can be "row", "col", "cell", or a vector of numerical indices.
- `rowMajor`: If TRUE, the first index varies fastest, if FALSE, the last index varies fastest.
- `drop`: Whether to drop marginalized dimensions. If chunking is used, this has no effect.
Details

This function is intended to follow the convention used in Python’s enumerate function where the primary difference is that a list is returned instead of Python’s tuple construct.

Each call to nextElem returns a list with two elements:

**index:** a counter

**value:** the current value of object

i_enum is an alias to i_enumerate to save a few keystrokes.

First appeared in package iterators2.

These are two closely closely related functions: i_enumerate accepts an iterable, and will only emit a single index starting with 1. i_enumerate is a generic with methods for vectors and arrays, supporting all chunking and recycling options, and returning multiple indices for arrays.

Value

iterator that returns the values of obj along with the index of the object.

Author(s)

Peter Meilstrup

Examples

```r
set.seed(42)
it <- i_enumerate(rnorm(5))
as.list(it)

# Iterates through the columns of the iris data.frame
it2 <- i_enum(iris)
nextOr(it2, NA)
nextOr(it2, NA)
nextOr(it2, NA)
nextOr(it2, NA)
nextOr(it2, NA)

a <- array(1:27, c(3, 3, 3))
as.list(i_enumerate(a, by=c(1, 2), drop=TRUE))
as.list(i_enumerate(a, by=c(3), drop=FALSE))
as.list(i_enumerate(a, by=c(2, 3), chunkSize=7))
```
**i_keep**  
*Iterator that filters elements not satisfying a predicate function*

**Description**

`i_keep(iterable, predicate)` constructs an iterator that filters elements from `iterable` returning only those for which the predicate is `TRUE`.

**Usage**

```r
i_keep(iterable, predicate, ...)
```

```r
i_drop(iterable, predicate, ...)
```

**Arguments**

- **iterable**: an iterable object.
- **predicate**: a function that determines whether an element is `TRUE` or `FALSE`. The function is assumed to take only one argument.
- **...**: passed along to `iteror` constructor.

**Details**

Originally called 'ifilter' from package `itertools`. Renamed because the order of arguments has changed to put the iterable in the first argument, the better to be used with the ` |> ` operator.

**Value**

iterator object

**See Also**

`i_drop` `i_keep` `while i_dropwhile`

**Examples**

```r
# Filters out odd numbers and retains only even numbers
is_even <- function(x) {
  x %% 2 == 0
}

it <- i_keep(1:10, is_even)
as.list(it)

# Similar idea here but anonymous function is used to retain only odd numbers
it2 <- i_drop(1:10, function(x) x %% 2 == 0)
nextOr(it2, NA) # 1
nextOr(it2, NA) # 3
```
i_keepwhile

Description

Constructs an iterator that returns elements from an iterable object as long as the given predicate function returns TRUE.

Usage

i_keepwhile(object, predicate, ...)

Arguments

object an iterable object
predicate a function that determines whether an element is TRUE or FALSE. The function is assumed to take only one argument.

... passed along to iteror(object, ...)
\texttt{i\_limit} \hspace{1cm} \textit{Limit the length of an iterator.}

\textbf{Value}

iterator object

\textbf{Examples}

\begin{verbatim}
# Filters out numbers exceeding 5
not\_too\_large <- function(x) {
  x <= 5
}
it <- i\_keepwhile(1:100, not\_too\_large)
unlist(as\_list(it)) == 1:5

# Same approach but uses an anonymous function
it2 <- i\_keepwhile(seq(2, 100, by=2), function(x) x <= 10)
unlist(as\_list(it2)) == c(2, 4, 6, 8, 10)
\end{verbatim}

\textbf{Description}

Create an iterator that limits the specified iterable to a specified number of items.

\textbf{Usage}

\begin{verbatim}
i\_limit(iterator, n, ...)
\end{verbatim}

\textbf{Arguments}

\begin{itemize}
  \item \texttt{iterable} \hspace{1cm} Iterable to iterate over.
  \item \texttt{n} \hspace{1cm} Maximum number of values to return.
  \item \texttt{...} \hspace{1cm} Extra arguments for \texttt{iteror(iterator, ...)}
\end{itemize}

\textbf{Details}

Originally from the \texttt{itertools} package.

\textbf{Value}

an \texttt{iteror} which will stop after yielding \texttt{n} values.

\textbf{Examples}

\begin{verbatim}
# Limit icount to only return three values
as\_list(i\_limit(icount(), 3))
\end{verbatim}
**i_map**

**Iterator that applies a given function to several iterables concurrently.**

**Description**

Constructs an iterator that computes the given function \( f \) using the arguments from each of the iterables given in \( \ldots \).

**Usage**

\[
i\_map(f, \ldots)
\]

**Arguments**

- \( f \) a function
- \( \ldots \) multiple arguments to iterate through in sequence

**Details**

The iterator returned is exhausted when the shortest iterable in \( \ldots \) is exhausted. Note that \( \text{i\_map} \) does not recycle arguments as Map does.

The primary difference between \( \text{i\_starmap} \) and \( \text{i\_map} \) is that the former expects an iterable object whose elements are already grouped together, while the latter case groups the arguments together before applying the given function. The choice is a matter of style and convenience.

**Value**

iterator that returns the values of object along with the index of the object.

**Examples**

\[
pow \leftarrow \text{function}(x, y) \{ \\
    x^y \\
\}
\]

\[
\text{it} \leftarrow \text{i\_map(pow, c(2, 3, 10), c(5, 2, 3))}
\]

\[
\text{as.list(it)}
\]

\[
\# \text{Similar to the above, but because the second vector is exhausted after two}
\# \text{calls to 'nextElem', the iterator is exhausted.}
\text{it2} \leftarrow \text{i\_map(pow, c(2, 3, 10), c(5, 2))}
\text{as.list(it2)}
\]

\[
\# \text{Another similar example but with lists instead of vectors}
\text{it3} \leftarrow \text{i\_map(pow, list(2, 3, 10), list(5, 2, 3))}
\text{nextOr(it3, NA)} \# 32
\text{nextOr(it3, NA)} \# 9
\text{nextOr(it3, NA)} \# 1000
\]
**i_mask**

Iterator that filters elements where corresponding selector is false.

**Description**

Constructs an iterator that filters elements from iterable returning only those for which the corresponding element from selectors is TRUE.

**Usage**

```r
i_mask(object, selectors)
```

**Arguments**

- **object** an iterable object
- **selectors** an iterable that determines whether the corresponding element in object is returned.

**Details**

The iterator stops when either object or selectors has been exhausted.

**Value**

iterator object

**Examples**

```r
# Filters out odd numbers and retains only even numbers
n <- 10
selectors <- rep(c(FALSE, TRUE), n)
it <- i_mask(seq_len(n), selectors)
as.list(it)

# Similar idea here but anonymous function is used to filter out even
# numbers
n <- 10
it2 <- i_mask(seq_len(10), rep(c(TRUE, FALSE), n))
as.list(it2)

it3 <- i_mask(letters, letters %in% c('a', 'e', 'i', 'o', 'u'))
as.list(it3)
```
i_pad

Iterator that returns an object followed indefinitely by a fill value

Description

Constructs an iterator that returns an iterable object before padding the iterator with the given fill value indefinitely.

Usage

i_pad(object, fill = NA, ...)

Arguments

object an iterable object
fill the value to pad the indefinite iterator after the initial object is consumed. Default: NA
...
... Passed along to iteror constructor.

Value

iterator that returns object followed indefinitely by the fill value

Examples

```r
it <- iteror(1:9)
it_i_pad <- i_pad(it)
as.list(i_slice(it_i_pad, end=9)) # Same as as.list(1:9)

it2 <- iteror(1:9)
it2_i_pad <- i_pad(it2)
as.list(i_slice(it2_i_pad, end=10)) # Same as as.list(c(1:9, NA))

it3 <- iteror(1:9)
it3_i_pad <- i_pad(it3, fill=TRUE)
as.list(i_slice(it3_i_pad, end=10)) # Same as as.list(c(1:9, TRUE))
```
**Description**

Create an iterator that recycles a specified iterable. On the first repeat the iterable is buffered into memory until it finishes, then we repeat the same sequence of values.

**Usage**

```r
i_recycle(iterable, times = Inf, ...)
```

**Arguments**

- `iterable`: The iterable to recycle.
- `times`: integer. Number of times to recycle the values. Default value of `Inf` means to recycle indefinitely.
- `...`: Further arguments will be passed along to `iteror`.

**Details**

Originally from the `itertools` package.

**Value**

an `iteror` recycling the values from the underlying iterable.

**Examples**

```r
# Recycle over 'a', 'b', and 'c' three times
i <- i_recycle(letters[1:3], 3)
as.character(i)

it <- i_recycle(1:3)
nextOr(it, NA) # 1
nextOr(it, NA) # 2
nextOr(it, NA) # 3
nextOr(it, NA) # 1
nextOr(it, NA) # 2
nextOr(it, NA) # 3
nextOr(it, NA) # 1

it2 <- i_recycle(1:3, times=2)
as.list(it2)
```
i_rep

Repeat values from an iterator.

Description

An analogue of the rep function operating on iterables.

Usage

i_rep(iterable, times = 1, length.out = NULL, each = 1, ...)

Arguments

- **iterable**: The iterable to iterate over repeatedly.
- **times**: How many times to recycle the underlying iteror (via i_recycle).
- **length.out**: The maximum length of output. If this is given times is ignored.
- **each**: The number of times to repeat each element. You can pass a vector (recycled), or another iterable, to repeat each element a varying number of times.
- **...**: further arguments passed along to iteror(iterable, ...)

Details

Note that arguments times and each can work slightly differently from rep; times must always be of length 1; to repeat each element a specific number of times, provide a vector to each rather than times.

Originally from the itertools package.

Value

an iteror yilding and repeating values from iterable.

See Also

base::rep, i_recycle

Examples

```
as.numeric(i_rep(1:4, 2))
as.numeric(i_rep(1:4, each=2))
as.numeric(i_rep(1:4, each=c(2,2,2,2)))
as.numeric(i_rep(1:4, each=c(2,1,2,1)))
as.numeric(i_rep(1:4, each=2, len=4))
as.numeric(i_rep(1:4, each=2, len=10))
as.numeric(i_rep(1:4, each=2, times=3))
```

# Note `rep` makes `times` behave like `each` when given a vector.
## i_repeat

Create a repeating iterator

### Description

Create an iterator that returns a value a specified number of times.

### Usage

```r
i_repeat(x, times)
```

### Arguments

- `x`  
The value to return repeatedly.
- `times`  
The number of times to repeat the value. Default value is infinity.

### Details

Originally from the `itertools` package.

### Value

an `iteror`.

### Examples

```r
# Repeat a value 10 times
unlist(as.list(i_repeat(42, 10)))
```
**Description**

This is an iterator equivalent of `rle`; it produces one output value for each run if identical values in its input, along with the length of the run. `i_rle_inverse()` performs the inverse transformation.

**Usage**

```
i_rle(obj, cmp = identical, ...)
i_rleinv(obj, ...)
```

**Arguments**

- `obj` An iterable
- `cmp` A function to use for comparison. It should take two arguments and return TRUE or FALSE.
- `...` Further arguments forwarded to `iteror(obj, ...)`. 

**Value**

An iterator returning entries of the form `list(length=n, value=X)`.

`i_rleinv` recreates the original data from the output of `i_rle`.

**Author(s)**

Peter Meilstrup

**See Also**

`i_dedupe`

**Examples**

```
it <- isample(c(TRUE, FALSE), 1, replace=TRUE)
rl <- i_rle(it)
x <- take(rl, 10)
as.logical(i_rleinv(x))
```
**i_roundrobin**  
*Iteror that traverses each given iterable in a roundrobin order*

**Description**

Constructs an iterator that traverses each given iterable in a roundrobin order. That is, the iterables are traversed in an alternating fashion such that the each element is drawn from the next iterable. If an iterable has no more available elements, it is skipped, and the next element is taken from the next iterable having available elements.

**Usage**

```r
i_roundrobin(...)  
```

**Arguments**

...  
multiple arguments to iterate through in roundrobin sequence

**Value**

iterator that alternates through each argument in roundrobin sequence

**Examples**

```r
it <- iteror(c("A", "B", "C"))  
it2 <- iteror("D")  
it3 <- iteror(c("E", "F"))  
as.list(i_roundrobin(it, it2, it3)) # A D E B F C
```

```r
it_rr <- i_roundrobin(1:3, 4:5, 7:10)  
as.list(it_rr) # 1 4 7 2 5 8 3 9 10
```

---

**i_slice**  
*Iteror that returns selected elements from an iterable.*

**Description**

Constructs an iteror that returns elements from an iterable following the given sequence with starting value `start` and ending value `end`. The sequence’s step size is given by `step`.

**Usage**

```r
i_slice(object, start = 1, end = NULL, step = 1, ...)
```
Arguments

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>object</td>
<td>iterable object through which this function iterates</td>
</tr>
<tr>
<td>start</td>
<td>the index of the first element to return from object</td>
</tr>
<tr>
<td>end</td>
<td>the index of the last element to return from object</td>
</tr>
<tr>
<td>step</td>
<td>the step size of the sequence</td>
</tr>
<tr>
<td>...</td>
<td>passed along to iteror(object, ...)</td>
</tr>
</tbody>
</table>

Details

The iterable given in object is traversed beginning with element having index specified in start. If start is greater than 1, then elements from the object are skipped until start is reached. By default, elements are returned consecutively. However, if the step size is greater than 1, elements in object are skipped.

If stop is Inf (default), the iteration continues until the iteror is exhausted unless end is specified. In this case, end specifies the sequence position to stop iteration.

Originally from package itertools2.

Value

iteror that returns object in sequence

Examples

```r
it <- i_slice(1:5, start=2)
nextOr(it, NULL) # 2
nextOr(it, NULL) # 3
nextOr(it, NULL) # 4
nextOr(it, NULL) # 5

it2 <- i_slice(1:10, start=2, end=5)
unlist(as.list(it2)) == 2:5

it3 <- i_slice(1:10, start=2, end=9, step=2)
unlist(as.list(it3)) == c(2, 4, 6, 8)
```

### i_starmap

**Iteror that applies a given function to the elements of an iterable.**

Description

Constructs an iteror that applies the function f concurrently to the elements within the list x.

Usage

```r
i_starmap(f, x)
i_star(f, x)
```
Arguments

- \( f \)   a function to apply to the elements of \( x \)
- \( x \) an iterable object

Details

The iterator returned is exhausted when the shortest element in \( x \) is exhausted. Note that \texttt{i_starmap} does not recycle arguments as \texttt{Map} does.

The primary difference between \texttt{i_starmap} and \texttt{i_map} is that the former expects an iterable object whose elements are already grouped together, while the latter case groups the arguments together before applying the given function. The choice is a matter of style and convenience.

Value

iterator that returns the values of object along with the index of the object.

Examples

```r
pow <- function(x, y) {
  x^y
}
it <- i_starmap(pow, list(c(2, 3, 10), c(5, 2, 3)))
unlist(as.list(it)) == c(32, 9, 1000)

# Similar to the above, but because the second vector is exhausted after two
# calls to \'nextElem\', the iterator is exhausted.
it2 <- i_starmap(pow, list(c(2, 3, 10), c(5, 2)))
unlist(as.list(it2)) == c(32, 9)

# Another similar example but with lists instead of vectors
it3 <- i_starmap(pow, list(list(2, 3, 10), list(5, 2, 3)))
as.list(it3)

# Computes sum of each row in the iris data set
# Numerically equivalent to base::rowSums()
tolerance <- sqrt(.Machine$double.eps)
iris_x <- iris[, -5]
it4 <- i_starmap(sum, iris_x)
unlist(as.list(it4)) - rowSums(iris_x) < tolerance
```

\textbf{i_tee}

Create multiple iterators from one source

Description

\texttt{i_tee(obj, n)} consumes and buffers the output of a single iterator \texttt{obj} so that it can be read by \texttt{n} independent sub-iterators.
Usage

\[
i_{\text{tee}}(\text{obj}, n, \text{max} = 2^{16} - 1, ...)\]

Arguments

- \text{obj} an iterable object
- \text{n} the number of iterators to return
- \text{max} The maximum number of values to buffer.
- \text{...} passed along to \text{iteror(obj, ...)}

Details

It works by saving the output of source \text{obj} in a queue, while each sub-iterator has a "read pointer" indexing into the queue. Items are dropped from the queue after all sub-iterations have seen them.

This means that if one sub-iterator falls far behind the others, or equivalently if one sub-iterator reads far ahead of its cohort the others, the intervening values will be kept in memory. The \text{max} argument gives a limit on how many items will be held. If this limit is exceeded due to one sub-iterator reading far ahead of the others, an error will be thrown when that sub-iterator attempts to read a new value.

Value

a list of \text{n} iterators.

Author(s)

Peter Meilstrup

Description

Create a timeout iterator

Create an iterator that iterates over another iterator for a specified period of time, and then stops. This can be useful when you want to search for something, or run a test for awhile, and then stop.

Usage

\[
i_{\text{timeout}}(\text{iterable}, \text{time}, ...)\]

Arguments

- \text{iterable} Iterable to iterate over.
- \text{time} The time interval to iterate for, in seconds.
- \text{...} passed along to \text{iteror(iterable, ...)}
i_unique

Details

Originally from the itertools package.

Value

an iterator yielding values from iterable so long as time is in the future

Examples

# See how high we can count in a tenth of a second
length(as.list(i_timeout(icount(), 0.1)))

i_unique

Iterator that extracts the unique elements from an iterable object

Description

Constructs an iterator that extracts each unique element in turn from an iterable object. Order of the elements is maintained. This function is an iterator analogue to unique.

Usage

i_unique(object, digest = rlang::hash, ...)

Arguments

object an iterable object
digest Optionally specify a custom hash function (e.g. digest::digest, rlang::hash).
It should be a function returning a character value.
... Extra arguments are forwarded to iteror.

Details

NOTE: In order to determine whether an element is unique, a list of previous unique elements is stored. In doing so, the list can potentially become large if there are a large number of unique elements.

Value

an iterator that returns only the unique elements from object

See Also

i_dedupe
**Examples**

```r
it <- i_chain(rep(1, 4), rep(2, 5), 4:7, 2)
as.list(i_unique(it)) # 1 2 4 5 6 7

it2 <- iterators::iter(c('a', 'a', 'A', 'V'))
as.list(i_unique(it2)) # a A V

x <- as.character(gl(5, 10))
it_unique <- i_unique(x)
as.list(it_unique) # 1 2 3 4 5
```

**Description**

Each element returned by `i_window(obj)` consists of `n` consecutive elements from the underlying `obj`, with the window advancing forward by one element each iteration.

**Usage**

```r
i_window(obj, n, tail, ...)
```

**Arguments**

- `obj` An iterable.
- `n` The width of the window to apply.
- `tail` If a value is given, tails will be included at the beginning and end of iteration, filled with the given value.
- `...` Passed along to `iteror(object, ...)`

**Value**

An iteror.

**Author(s)**

Peter Meilstrup

**Examples**

```r
# @examples
it <- i_window(iteror(letters[1:4]), 2)
nextOr(it, NA) # list("a", "b")
nextOr(it, NA) # list("b", "c")
nextOr(it, NA) # list("c", "d")
```
```
itz <- i_window(icount(5), 2)
nextOr(itz, NA) # list(1, 2)
nextOr(itz, NA) # list(2, 3)
nextOr(itz, NA) # list(3, 4)
nextOr(itz, NA) # list(4, 5)

itz <- i_window(letters[1:4], 2)
nextOr(itz, NA) # list("a", "b")
nextOr(itz, NA) # list("b", "c")
nextOr(itz, NA) # list("c", "d")

itz <- i_window(letters[1:4], 3)
nextOr(itz) # list("a", "b", "c")
nextOr(itz) # list("b", "c", "d")

itz <- i_window(letters[1:4], 3, tail=" ")
nextOr(itz) # list( "", "", "a")
nextOr(itz) # list( "", "a", "b")
nextOr(itz) # list( "a", "b", "c")
nextOr(itz) # list( "b", "c", "d")
nextOr(itz) # list( "c", "d", " ")
nextOr(itz) # list( "d", " ", " ")
```

---

**i_zip**  
*Combine several iterables in parallel.*

---

**Description**

The resulting iterator aggregates one element from each of the iterables into a list for each iteration. Used for lock-step iteration over several iterables at a time.

**Usage**

```
i_zip(...)  
i_zip_longest(..., fill = NA)
```

**Arguments**

- `...` multiple arguments to iterate through in parallel
- `fill` the value used to replace missing values when the iterables in ... are of uneven length

**Details**

For `i_zip`, the output will finish when any of the underlying iterables finish.

Originally from the `itertools` package.

Originally from package `itertools2`.
makeIwrapper

Iterator Constructor-Constructor Function Wrapper

Description

The `makeIwrapper` function wraps an R function to produce an iterator constructor. It is used to construct random sampling iterators in this package; for instance `irnorm` is defined as `irnorm <- makeIwrapper(rnorm)`.

Usage

```
makeIwrapper(FUN)
```

Arguments

- **FUN**: a function that generates different values each time it is called; typically one of the standard random number generator functions.

Examples

# Iterate over two iterables of different sizes
as.list(i_zip(a=1:2, b=letters[1:3]))

it <- i_zip_longest(x=1:3, y=4:6, z=7:9)
nextOr(it, NA) # list(x=1, y=4, z=7)
nextOr(it, NA) # list(x=2, y=5, z=8)
nextOr(it, NA) # list(x=3, y=6, z=9)

it2 <- i_zip_longest(1:3, 4:8)
nextOr(it2, NA) # list(1, 4)
nextOr(it2, NA) # list(2, 5)
nextOr(it2, NA) # list(3, 6)
nextOr(it2, NA) # list(NA, 7)
nextOr(it2, NA) # list(NA, 8)

it3 <- i_zip_longest(1:2, 4:7, levels(iris$Species), fill="w00t")
nextOr(it3, NA) # list(1, 4, "setosa")
nextOr(it3, NA) # list(2, 5, "versicolor")
nextOr(it3, NA) # list("w00t", 6, "virginica")
nextOr(it3, NA) # list("w00t", 7, "w00t")
Details

The resulting iterator constructors all take an optional count argument which specifies the number of times the resulting iterator should fire. They also have an argument independent which enables independent tracking of the random number seed. The isample function is an example of one such iterator constructoe (as are irnorm, irunif, etc.).

Original version appeared in the iterators package.

Value

An iterator that is a wrapper around the corresponding function.

Examples

```r
# create an iterator maker for the sample function
mysample <- makeIwrapper(sample)
# use this iterator maker to generate an iterator that will generate three five
# member samples from the sequence 1:100
it <- mysample(1:100, 5, count = 3)
extOr(it)
extOr(it)
extOr(it)
extOr(it, NULL)  # NULL
```

nextOr  Retreive the next element from an iteor.

Description

Retreive the next element from an iteor.

Usage

`nextOr(obj, or, ...)`

Arguments

- `obj`  
  An iterator.
- `or`  
  If the iterator has reached its end, this argument will be forced and returned.
- `...`  
  Other arguments may be used by specific iteors.

Value

Either the next value of iterator, or the value of or.
nth

Returns the nth item of an iterator

Description

Returns the nth item of an iterator after advancing the iterator n steps ahead. If the iterator is entirely consumed, the argument or is returned instead. That is, if either \( n > \text{length(iterator)} \) or \( n \) is 0, then the iterator is consumed.

Usage

\[
nth(obj, n, \text{or}, \ldots)
\]

Arguments

- **obj**: an iterable.
- **n**: The index of the desired element to return.
- **or**: If the iterator finishes before returning \( n \) elements, this argument will be forced and returned.
- **...**: passed along to iterator constructor.

Value

The nth element of the iterator or the result of forcing or.

See Also

take consume collect

Examples

\[
\text{it} \leftarrow \text{iteror(1:10)}
# Returns 5
nth(it, 5, \text{NA})
\]

\[
\text{it2} \leftarrow \text{iteror(letters)}
# Returns 'e'
nth(it2, 5, \text{NA})
\]

\[
\text{it3} \leftarrow \text{iteror(letters)}
# Returns default value of NA
nth(it3, 42, \text{NA})
\]

\[
\text{it4} \leftarrow \text{iteror(letters)}
# Returns default value of "foo"
nth(it4, 42, \text{or}="foo")
\]
**quantify**

*Count the number of times an iterable object is TRUE*

**Description**

Returns the number of elements from an iterable object that evaluate to TRUE.

**Usage**

```r
quantify(obj, ...)
```

**Arguments**

- `obj`: an iterable object
- `...`: further arguments passed to `iteror`.

**Value**

the number of TRUE elements

**See Also**

`reduce`

**Examples**

```r
it <- iteror(c(TRUE, FALSE, TRUE))
quantify(it) # 2

set.seed(42)
x <- sample(c(TRUE, FALSE), size=10, replace=TRUE)
quantify(x) # Equivalent to sum(x)
```

---

**record**

*Record and replay iterators*

**Description**

The `record` function records the values issued by a specified iterator to a file or connection object. The `ireplay` function returns an iterator that will replay those values. This is useful for iterating concurrently over multiple, large matrices or data frames that you can’t keep in memory at the same time. These large objects can be recorded to files one at a time, and then be replayed concurrently using minimal memory.

**Usage**

```r
record(iterable, con, ...)
```
Arguments

iterable  The iterable to record to the file.
con       A file path or open connection.
...       passed along to itor(iterable, ...)

Details

Originally from the itertools package.

Value

NULL, invisibly.

Examples

suppressMessages(library(foreach))
m1 <- matrix(rnorm(70), 7, 10)
f1 <- tempfile()
record(itor(m1, by='row', chunkSize=3), f1)

m2 <- matrix(1:50, 10, 5)
f2 <- tempfile()
record(itor(m2, by='column', chunkSize=3), f2)

# Perform a simple out-of-core matrix multiply
p <- foreach(col=ireplay(f2), .combine='cbind') %:%
   foreach(row=ireplay(f1), .combine='rbind') %do% {
      row %*% col
   }
dimnames(p) <- NULL
print(p)
all.equal(p, m1 %*% m2)
unlink(c(f1, f2))

reduce  Compute the sum, product, or general reduction of an iterator.

Description

reduce(obj, fun) applies a 2-argument function fun between successive elements of obj. For example if fun is +, reduce(it, +, init=0) computes 0 + nextElem(it) + nextElem(it) + nextElem(it) + ... until the iterator finishes, and returns the final value.

i_accum(obj) returns the iterator containing each intermediate result. The default settings produce a cumulative sum.
sum.iteror(it) is equivalent to reduce(it, `+`)  
prod.iteror(it) is equivalent to reduce(it, `*`).

Usage

reduce(obj, fun = `+`, init = 0, ...)

## S3 method for class 'iteror'
reduce(obj, fun = `+`, init = 0, ...)

i_accum(obj, fun = `+`, init = 0, ...)

## S3 method for class 'iteror'
sum(..., na.rm = FALSE)

## S3 method for class 'iteror'
prod(..., na.rm = FALSE)

Arguments

obj
an iterable object

fun
A function of at least two arguments.

init
A starting value.

...    Extra parameters will be passed to each call to fun.

na.rm
Whether to drop NA values when computing sum or prod.

Value

The result of accumulation.

Author(s)

Peter Meilstrup

Examples

it <- icount(5)
total <- reduce(it, `+`) # 15

it <- icount(5)
reduce(it, paste0, "") # "12345"

it <- icount(5)
reduce(it, `*`, init=1) # 120

# triangular numbers: 1, 1+2, 1+2+3, ...
take(i_accum(icount()), 10, 'numeric')
r_to_py.iteror  
Wrap an iteror to appear as a Python iterator or vice versa.

Description
This requires the reticulate package to be installed.

Usage

```r
## S3 method for class 'iteror'
r_to_py(x, convert = FALSE, ...)

## S3 method for class 'python.builtin.object'
iteror(obj, ...)
```

Arguments

- **x**  An iterable object.
- **convert**  does nothing.
- **...**  Passed along to `iteror(x, ...)`.  
- **obj**  A Python object (as viewed by package reticulate.)

Value

`r_to_py(it)` returns a Python iterator.

Method `iteror.python.builtin.object` returns an `iteror`.

Examples

```r
pit <- reticulate::r_to_py(iseq(2, 11, 5))
reticulate::iter_next(pit, NULL)
reticulate::iter_next(pit, NULL)
reticulate::iter_next(pit, NULL)

# create an R iterator and ask Python to sum it  
triangulars <- icount |> i_accum() |> i_limit(10)  
builtins <- reticulate::import_builtins()  
builtins$sum(triangulars)  # r_to_py is called automatically

# create a generator in Python and sum it in R  
pit <- reticulate::py_eval("(n for n in range(1, 25) if n % 3 == 0")  
sum(iteror(pit))
```
Description

Returns the first \( n \) elements of an iterable object as a list. If \( n \) is larger than the number of elements in object, the entire iterator is consumed.

Usage

\[
\text{take(obj, } n = 1, \text{ mode = "list", ...)}
\]

## Default S3 method:
\[
\text{take(obj, } n = 1, \text{ mode = "list", ...)}
\]

## S3 method for class 'iteror'
\[
\text{take(obj, } n = 1, \text{ mode = "list", ...)}
\]

Arguments

- **obj**: An iterable object.
- **n**: The maximum number of elements to extract from the iteror.
- **mode**: The mode of vector to return.
- **...**: Further arguments may be passed along to the iteror constructor.

Details

A function `take` first appeared in package `itertools`. It is basically an alias for `as.list` but defaults to \( n=1 \).

Value

A list of the first \( n \) items of the iterable `obj`.

See Also

`concat`, `as.vector.iteror`, `as.vector.iteror`

Examples

\[
\text{take(1:10, 3)} \ # \ 1 \ 2 \ 3
\]
\[
\text{take(icount(), 10)} \ # \ 1:10
\]
\[
\text{take(icount(5), 10)} \ # \ 1 \ 2 \ 3 \ 4 \ 5
\]
Index

* methods
  hasNext, 7
* utilities
  i_rep, 46
  .Random.seed, 22
  as.character.iteror (as.list.iteror), 3
  as.double.iteror (as.list.iteror), 3
  as.list, 63
  as.list.iteror, 3
  as.logical.iteror (as.list.iteror), 3
  as.numeric.iteror (as.list.iteror), 3
  as.vector.iteror (as.list.iteror), 3
  base::rep, 46
  c(), 35
  combn, 8
  concat, 4
  consume, 5
  count, 6
  data.frame, 18
  dotproduct, 7
  expand.grid, 14
  hasNext, 7
  i_accum (reduce), 60
  i_apply, 32
  i_break, 33
  i_chain (i_concat), 35
  i_chunk, 34
  i_concat, 35
  i_drop (i_keep), 39
  i_dropwhile, 36
  i_enum (iEnumerate), 37
  iEnumerate, 30, 37
  i_keep, 39
  i_keepwhile, 40
  i_limit, 41
  i_map, 33, 42, 42, 51
  i_mask, 43
  i_pad, 44
  i_recycle, 45, 46
  i_repeat, 46
  i_repeat, 47
  i_rle, 48
  i_rleinv (i_rle), 48
  i_roundrobin, 49
  i_slice, 49
  i_start (i_starmap), 50
  i_starmap, 50
  i_tee, 51
  i_timeout, 52
  i_unique, 53
  i_window, 54
  i_zip, 55
  i_zip_longest (i_zip), 55
  icombinations, 8
  icount, 10
  icountn (icount), 10
  idedup, 11
  identical, 29
  identical(), 31
  idiv, 12
  iEnumerate (iEnumerate), 37
  ifilter (i_keep), 39
  ifilterfalse (i_keep), 39
  igrid, 13
  ihasNext (hasNext), 7
  ipermutations, 14
  irbinom (irnorm), 20
  iread.table, 15
  ireadBin, 16
  ireaddf, 17
  ireadLines, 18
  ireplay (record), 59
  irnbinom (irnorm), 20
INDEX

iRNGStream, 19, 22, 23
iRNGSubStream(iRNGStream), 19
irnorm, 20
irpois(irnorm), 20
is.iterator(is.iteror), 24
is.iteror, 24
isample, 23
isample(irnorm), 20
iseq, 10, 24
iseq_along(iseq), 24
isplit, 26
itabulate, 27
iterators::iter, 29
iterators::nextElem, 8
iteror, 4–8, 14, 17–19, 25, 27, 31, 35, 36, 39, 41, 44, 45, 47, 53, 57–59, 62, 63
iteror.array, 29
iteror.data.frame(iteror.array), 29
iteror.function, 29, 31
iteror.matrix(iteror.array), 29
iteror.pythonbuiltin.object
  (r_to_py.iteror), 62
makeIwrapper, 56
Map, 42, 51

nextElem, 38
nextOr, 8, 57
nextRNGStream, 19
nextRNGSubStream, 19
nth, 58

parallel::nextRNGStream, 19
prod.iteror(reduce), 60
quantify, 59

r_to_py.iteror, 62
rbinom, 22
read.table, 16
readLines, 18
record, 59
reduce, 60
rep, 46
rle, 48
rnbinom, 22
rnorm, 22
rpois, 22