

Package ‘itertools2’

July 22, 2025

Title itertools2: Functions creating iterators for efficient looping

Version 0.1.1

Date 2014-08-08

Author John A. Ramey <johnramey@gmail.com>,
Kayla Schaefer <kschaefer.tx@gmail.com>

Maintainer John A. Ramey <johnramey@gmail.com>

Description A port of Python's excellent itertools module to R for efficient looping.

Depends R (>= 3.0.2)

Imports iterators (>= 1.0.7)

Suggests testthat (>= 0.8.1)

License MIT + file LICENSE

URL <https://github.com/ramhiser/itertools2>, <http://ramhiser.com>

NeedsCompilation no

Repository CRAN

Date/Publication 2014-08-08 15:16:57

Contents

consume	2
dotproduct	3
ichain	4
ichunk	4
icompress	5
icount	6
icycle	7
idropwhile	8
ienumerate	9
ifilter	10
ilength	11
imap	12

ipad	13
ipairwise	14
iproduct	14
irep	15
irepeat	16
iroundrobin	17
iseq	18
islice	19
istarmap	20
is_iterator	21
itabulate	22
itakewhile	22
itee	23
itertools2	24
iter_deepcopy	24
iter_length	25
iunique	25
iunique_justseen	26
izip	27
izip_longest	28
nth	29
quantify	30
stop_iteration	30
take	31
try_nextElem	31

Index 33

consume	<i>Consumes the first n elements of an iterator</i>
---------	---

Description

Advances the iterator n-steps ahead without returning anything.

Usage

```
consume(iterator, n = 0)
```

Arguments

iterator	an iterator object
n	The number of elements to consume.

Details

If n is 0, the iterator is consumed entirely. Similarly, if n is larger than the length of the iterator, the iterator is consumed entirely.

Value

Nothing, i.e., invisible(NULL)

Examples

```
it <- iterators::iter(1:10)
# Skips the first 5 elements
consume(it, n=5)
# Returns 6
iterators::nextElem(it)

it2 <- iterators::iter(letters)
# Skips the first 4 elements
consume(it2, 4)
# Returns 'e'
iterators::nextElem(it2)
```

dotproduct

Computes the dot product of two iterable objects

Description

Returns the dot product of two numeric iterables of equal length

Usage

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

Arguments

vec1	the first
vec2	the second iterable object

Value

the dot product of the iterators

Examples

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

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

ichain	<i>Iterator that chains multiple arguments together into a single iterator</i>
--------	--

Description

Generates an iterator that returns elements from the first argument until it is exhausted. Then generates an iterator from the next argument and returns elements from it. This process continues until all arguments are exhausted. Chaining is useful for treating consecutive sequences as a single sequence.

Usage

```
ichain(...)
```

Arguments

... multiple arguments to iterate through in sequence

Value

iterator that iterates through each argument in sequence

Examples

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

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

ichunk	<i>Iterator that returns elements in fixed-length chunks</i>
--------	--

Description

Constructs an iterator that returns elements of an iterable object in fixed-length chunks. If the length of the iterator is not divisible by `chunk_size`, the remainder of the last block is filled with the value specified in `fill`.

Usage

```
ichunk(object, chunk_size = 1, fill = NA)
```

Arguments

<code>object</code>	an iterable object
<code>chunk_size</code>	the number of elements returned per chunk
<code>fill</code>	the value with which to fill the last chunk if the length of the iterator is not divisible by <code>chunk_size</code>

Details

This function corresponds to Python's grouper function. We chose the name `ichunk` because it more explicitly defines the function's purpose.

Value

each call to `nextElem` results in a list of length `chunk_size`

Examples

```
it <- ichunk(iterators::iter(1:5), chunk_size=2)
# List: list(1, 2, 3)
iterators::nextElem(it)
# List: list(4, 5, NA)
iterators::nextElem(it)

it2 <- ichunk(levels(iris$Species), chunk_size=4, "weeee")
# Returns: list("setosa", "versicolor", "virginica", "weeee")
iterators::nextElem(it2)
```

`icompress`*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

```
icompress(object, selectors)
```

Arguments

<code>object</code>	an iterable object
<code>selectors</code>	an iterable that determines whether the corresponding element in <code>object</code> is returned.

Details

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

Value

iterator object

Examples

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

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

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

icount	<i>Iterator of neverending numeric sequence with initial value and step size</i>
--------	--

Description

Constructs an iterator that generates a neverending sequence of evenly spaced values starting with `icount`. The step size is given by `step`.

Usage

```
icount(start = 0, step = 1)
```

Arguments

<code>start</code>	sequence's initial value
<code>step</code>	sequence's step size

Details

NOTE: Use a negative step size to generate decreasing sequences.

Often used as an argument to [imap](#) to generate consecutive data points.

Value

sequence's iterator

Examples

```
it <- icount()
iterators::nextElem(it)
iterators::nextElem(it)
iterators::nextElem(it)

it2 <- icount(start=5.5, step=1.5)
iterators::nextElem(it2)
iterators::nextElem(it2)
iterators::nextElem(it2)
```

icycle*Iterator that cycles indefinitely through an iterable object*

Description

Constructs an iterator that returns an iterable object in sequence over and over again.

Usage

```
icycle(object, times = NULL)
```

Arguments

object	object to cycle indefinitely.
times	the number of times object is returned. If NULL (default), object is returned indefinitely.

Details

Runs indefinitely unless the `times` argument is specified.

Value

iterator that returns object in sequence

Examples

```
it <- icycle(1:3)
iterators::nextElem(it) # 1
iterators::nextElem(it) # 2
iterators::nextElem(it) # 3
iterators::nextElem(it) # 1
iterators::nextElem(it) # 2
iterators::nextElem(it) # 3
iterators::nextElem(it) # 1

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

```
# Can return the results from a function.
it3 <- icycle(function() rnorm(1))
iterators::nextElem(it)
iterators::nextElem(it)
iterators::nextElem(it)
iterators::nextElem(it)
```

idropwhile	<i>Iterator that drops elements until the predicate function returns FALSE</i>
------------	--

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

```
idropwhile(predicate, object)
```

Arguments

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

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

iterator object

Examples

```
# Filters out numbers exceeding 3
not_too_large <- function(x) {
  x <= 3
}
it <- idropwhile(not_too_large, 1:8)
as.list(it)

# Same approach but uses an anonymous function
it2 <- idropwhile(function(x) x <= 10, seq(2, 20, by=2))
as.list(it2)
```

`ienumerate`*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.

Usage

```
ienumerate(object)
```

```
ienum(object)
```

Arguments

`object` object to return indefinitely.

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

`ienum` is an alias to `ienumerate` to save a few keystrokes.

Value

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

Examples

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

# Iterates through the columns of the iris data.frame
it2 <- ienum(iris)
iterators::nextElem(it2)
iterators::nextElem(it2)
iterators::nextElem(it2)
iterators::nextElem(it2)
iterators::nextElem(it2)
```

`ifilter`*Iterator that filters elements not satisfying a predicate function*

Description

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

Constructs an iterator that filters elements from iterable returning only those for which the predicate is FALSE.

Usage

```
ifilter(predicate, iterable)
```

```
ifilterfalse(predicate, iterable)
```

Arguments

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

`iterable` an iterable object

Value

iterator object

Examples

```
# Filters out odd numbers and retains only even numbers
is_even <- function(x) {
  x %% 2 == 0
}
it <- ifilter(is_even, 1:10)
as.list(it)
```

```
# Similar idea here but anonymous function is used to filter out even
# numbers
it2 <- ifilter(function(x) x %% 2 == 1, 1:10)
iterators::nextElem(it2) # 1
iterators::nextElem(it2) # 3
iterators::nextElem(it2) # 5
iterators::nextElem(it2) # 7
iterators::nextElem(it2) # 9
```

```
is_vowel <- function(x) {
  x %in% c('a', 'e', 'i', 'o', 'u')
}
it3 <- ifilter(is_vowel, letters)
```

```
as.list(it3)
# Filters out even numbers and retains only odd numbers
is_even <- function(x) {
  x %% 2 == 0
}
it <- ifilterfalse(is_even, 1:10)
as.list(it)

# Similar idea here but anonymous function is used to filter out odd
# numbers
it2 <- ifilter(function(x) x %% 2 == 1, 1:10)
as.list(it2)

is_vowel <- function(x) {
  x %in% c('a', 'e', 'i', 'o', 'u')
}
it3 <- ifilterfalse(is_vowel, letters)
iterators::nextElem(it3) # b
iterators::nextElem(it3) # c
iterators::nextElem(it3) # d
iterators::nextElem(it3) # f
iterators::nextElem(it3) # g
# iterators::nextElem(it) continues through the rest of the consonants
```

ilength

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

```
ilength(object)
```

Arguments

object an iterable object

Value

the number of elements in the iterator

Examples

```
ilength(1:5) == length(1:5)

it <- iterators::iter(1:5)
ilength(it) == length(1:5)
```

```
it2 <- ichain(1:3, 4:5, 6)
ilength(it2)

it3 <- ichain(1:3, levels(iris$Species))
ilength(it3)
```

imap

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 `...`

Usage

```
imap(f, ...)
```

Arguments

<code>f</code>	a function
<code>...</code>	multiple arguments to iterate through in sequence

Details

The iterator returned is exhausted when the shortest iterable in `...` is exhausted. Note that `imap` does not recycle arguments as `Map` does.

The primary difference between `istarmap` and `imap` 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 <- function(x, y) {
  x^y
}
it <- imap(pow, c(2, 3, 10), c(5, 2, 3))
as.list(it)

# Similar to the above, but because the second vector is exhausted after two
# calls to `nextElem`, the iterator is exhausted.
it2 <- imap(pow, c(2, 3, 10), c(5, 2))
as.list(it2)
```

```
# Another similar example but with lists instead of vectors
it3 <- imap(pow, list(2, 3, 10), list(5, 2, 3))
iterators::nextElem(it3) # 32
iterators::nextElem(it3) # 9
iterators::nextElem(it3) # 1000
```

ipad

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

```
ipad(object, fill = NA)
```

Arguments

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

Value

iterator that returns object followed indefinitely by the fill value

Examples

```
it <- iterators::iter(1:9)
it_ipad <- ipad(it)
as.list(islice(it_ipad, end=9)) # Same as as.list(1:9)

it2 <- iterators::iter(1:9)
it2_ipad <- ipad(it2)
as.list(islice(it2_ipad, end=10)) # Same as as.list(c(1:9, NA))

it3 <- iterators::iter(1:9)
it3_ipad <- ipad(it3, fill=TRUE)
as.list(islice(it3_ipad, end=10)) # Same as as.list(c(1:9, TRUE))
```

ipairwise	<i>Iterator that returns elements of an object in pairs</i>
-----------	---

Description

Constructs an iterator of an iterable object that returns its elements in pairs.

Usage

```
ipairwise(object)
```

Arguments

object an iterable object

Value

an iterator that returns pairwise elements

Examples

```
it <- ipairwise(iterators::iter(letters[1:4]))
iterators::nextElem(it) # list("a", "b")
iterators::nextElem(it) # list("b", "c")
iterators::nextElem(it) # list("c", "d")

it2 <- ipairwise(1:5)
iterators::nextElem(it2) # list(1, 2)
iterators::nextElem(it2) # list(2, 3)
iterators::nextElem(it2) # list(3, 4)
iterators::nextElem(it2) # list(4, 5)
```

iproduct	<i>Iterator that returns the Cartesian product of the arguments.</i>
----------	--

Description

Constructs an iterator that is the Cartesian product of each of the arguments.

Usage

```
iproduct(...)
```

Arguments

... multiple arguments

Details

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

Value

iterator that iterates through each element from the Cartesian product

Examples

```
it <- iproduct(x=1:3, y=4:5)
iterators::nextElem(it) # list(x=1, y=4)
iterators::nextElem(it) # list(x=1, y=5)
iterators::nextElem(it) # list(x=2, y=4)
iterators::nextElem(it) # list(x=2, y=5)
iterators::nextElem(it) # list(x=3, y=4)
iterators::nextElem(it) # list(x=3, y=5)

# iproduct is a replacement for base::expand.grid()
# Large data.frames are not created unless the iterator is manually consumed
a <- 1:2
b <- 3:4
c <- 5:6
it2 <- iproduct(a=a, b=b, c=c)
df_iproduct <- do.call(rbind, as.list(it2))
df_iproduct <- data.frame(df_iproduct)

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

irep

Iterator that replicates elements of an iterable object

Description

Constructs an iterator that replicates the values of an object.

Usage

```
irep(object, times = 1, length.out = NULL, each = NULL)
```

```
irep_len(object, length.out = NULL)
```

Arguments

<code>object</code>	object to return indefinitely.
<code>times</code>	the number of times to repeat each element in object
<code>length.out</code>	non-negative integer. The desired length of the iterator
<code>each</code>	non-negative integer. Each element is repeated each times

Details

This function is intended an iterable version of the standard [rep](#) function. However, as exception, the recycling behavior of [rep](#) is intentionally not implemented.

Value

iterator that returns object

Examples

```
it <- irep(1:3, 2)
unlist(as.list(it)) == rep(1:3, 2)

it2 <- irep(1:3, each=2)
unlist(as.list(it2)) == rep(1:3, each=2)

it3 <- irep(1:3, each=2, length.out=4)
as.list(it3)
```

irepeat

Iterator that returns an object indefinitely

Description

Constructs an iterator that returns an object over and over again.

Usage

```
irepeat(object, times = NULL)
```

Arguments

object	object to return indefinitely.
times	the number of times object is returned. If NULL (default), object is returned indefinitely.

Details

Runs indefinitely unless the `times` argument is specified. Used as argument to [imap](#) for invariant function parameters. Also used with [izip](#) to create constant fields in a tuple record.

Value

iterator that returns object

Examples

```
it <- irepeat(42)
iterators::nextElem(it)
iterators::nextElem(it)
iterators::nextElem(it)
# Further calls to iterators::nextElem(it) will repeat 42

it2 <- irepeat(42, times=4)
iterators::nextElem(it2)
iterators::nextElem(it2)
iterators::nextElem(it2)
iterators::nextElem(it2)

# The object can be a data.frame, matrix, etc
it3 <- irepeat(iris, times=4)
iterators::nextElem(it3)
iterators::nextElem(it3)
iterators::nextElem(it3)
iterators::nextElem(it3)
```

iroundrobin

Iterator 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

```
iroundrobin(...)
```

Arguments

... multiple arguments to iterate through in roundrobin sequence

Value

iterator that alternates through each argument in roundrobin sequence

Examples

```
it <- iterators::iter(c("A", "B", "C"))
it2 <- iterators::iter("D")
it3 <- iterators::iter(c("E", "F"))
as.list(iroundrobin(it, it2, it3)) # A D E B F C
```

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

iseq

Iterators for sequence generation

Description

Constructs iterators that generate regular sequences that follow the [seq](#) family.

Usage

```
iseq(from = 1, to = 1, by = (to - from)/(length_out - 1),
     length_out = NULL, along_with = NULL)

iseq_len(length_out = NULL)

iseq_along(along_with = NULL)
```

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 <code>seq</code> will be rounded up if fractional.
along_with	the length of the sequence will match the length of this argument

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`.

`seq_along` and `seq_len` return an iterator, which generates a sequence of integers, beginning with 1 and proceeding to an ending value

Value

sequence's iterator

Examples

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

it2 <- iseq_len(4)
unlist(as.list(it2)) == 1:4

it3 <- iseq_along(iris)
unlist(as.list(it3)) == 1:length(iris)
```

islice

Iterator that returns selected elements from an iterable.

Description

Constructs an iterator 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

```
islice(object, start = 1, end = NULL, step = 1)
```

Arguments

<code>object</code>	iterable object through which this function iterates
<code>start</code>	the index of the first element to return from object
<code>end</code>	the index of the last element to return from object
<code>step</code>	the step size of the sequence

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 `step` is `NULL` (default), the iteration continues until the iterator is exhausted unless `end` is specified. In this case, `end` specifies the sequence position to stop iteration.

Value

iterator that returns object in sequence

Examples

```
it <- islice(1:5, start=2)
iterators::nextElem(it) # 2
iterators::nextElem(it) # 3
iterators::nextElem(it) # 4
iterators::nextElem(it) # 5

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

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

istarmap

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

Description

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

Usage

```
istarmap(f, x)
```

```
istar(f, x)
```

Arguments

<code>f</code>	a function to apply to the elements of <code>x</code>
<code>x</code>	an iterable object

Details

The iterator returned is exhausted when the shortest element in `x` is exhausted. Note that `istarmap` does not recycle arguments as `Map` does.

The primary difference between `istarmap` and `imap` 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 <- function(x, y) {
  x^y
}
it <- istarmap(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 <- istarmap(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 <- istarmap(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 <- istarmap(sum, iris_x)
unlist(as.list(it4)) - rowSums(iris_x) < tolerance
```

is_iterator

Helper function that determines whether is an iterator object

Description

Returns TRUE if the object is an object of class iter, and FALSE otherwise.

Usage

```
is_iterator(object)
```

Arguments

object an R object

Value

logical value indicating whether object is of class iter

itabulate	<i>Iterator that maps a function to a sequence of numeric values</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 `start` and are incremented by `step`.

Usage

```
itabulate(f, start = 1, step = 1)
```

Arguments

<code>f</code>	the function to apply
<code>start</code>	sequence's initial value
<code>step</code>	sequence's step size

Value

an iterator that returns the mapped values from the sequence

Examples

```
it <- itabulate(f=function(x) x + 1)
take(it, 4) # 2 3 4 5

it2 <- itabulate(f=function(x) x^2, start=-3)
take(it2, 6) # 9 4 1 0 1 4

it3 <- itabulate(abs, start=-5, step=2)
take(it3, 6) # 5 3 1 1 3 5

it4 <- itabulate(exp, start=6, step=-2)
take(it4, 4) # exp(c(6, 4, 2, 0))
```

itakewhile	<i>Iterator that returns elements while a predicate function returns TRUE</i>
------------	---

Description

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

Usage

```
itakewhile(predicate, object)
```

Arguments

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

object an iterable object

Value

iterator object

Examples

```
# Filters out numbers exceeding 5
not_too_large <- function(x) {
  x <= 5
}
it <- itakewhile(not_too_large, 1:100)
unlist(as.list(it)) == 1:5

# Same approach but uses an anonymous function
it2 <- itakewhile(function(x) x <= 10, seq(2, 100, by=2))
unlist(as.list(it2)) == c(2, 4, 6, 8, 10)
```

```
itee
```

Returns a list of n independent iterators from a single iterable object

Description

Constructs a list of n iterators, each of which iterates through an iterable object.

Usage

```
itee(object, n = 2)
```

Arguments

object an iterable object

n the number of iterables to return

Details

If the object is an iterator (i.e., inherits from class `iter`), n deep copies of object are returned. Otherwise, object is passed to `iter` n times.

Value

a list of n iterators

Examples

```
# Creates a list of three iterators.
# Each iterator iterates through 1:5 independently.
iter_list <- itee(1:5, n=3)

# Consumes the first iterator
unlist(as.list(iter_list[[1]])) == 1:5

# We can iterate through the remaining two iterators in any order.
iterators::nextElem(iter_list[[2]]) # 1
iterators::nextElem(iter_list[[2]]) # 2

iterators::nextElem(iter_list[[3]]) # 1
iterators::nextElem(iter_list[[3]]) # 2

iterators::nextElem(iter_list[[2]]) # 3
iterators::nextElem(iter_list[[2]]) # 4
iterators::nextElem(iter_list[[2]]) # 5

iterators::nextElem(iter_list[[3]]) # 3
iterators::nextElem(iter_list[[3]]) # 4
iterators::nextElem(iter_list[[3]]) # 5
```

itertools2

itertools2: Functions creating iterators for efficient looping

Description

The R package `itertools2` is a port of Python's excellent `itertools` module <https://docs.python.org/2/library/itertools.html> to R for efficient looping and is a replacement for the existing `itertools` R package <https://r-forge.r-project.org/projects/itertools/>.

iter_deepcopy

Performs a deep copy of an iterator

Description

This function is useful when an iterator needs to be copied with a new state environment.

Usage

```
iter_deepcopy(iterator)
```


Arguments

iterator an iterator object that inherits from class 'iter'

Value

a new iterator with its own state

iter_length *Helper function that determines the length of an iterator object*

Description

Returns the length of an iterator object. In the case that the iterator's length is NULL, a value of 1 is returned by default. This value can be set using the default argument.

Usage

```
iter_length(object, default = 1)
```

Arguments

object an iterator object
default the value returned when an iterator has NULL length

Value

integer

iunique *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 [sort](#).

Usage

```
iunique(object)
```

Arguments

object an iterable object

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 the unique elements from object

Examples

```
it <- ichain(rep(1, 4), rep(2, 5), 4:7, 2)
as.list(iunique(it)) # 1 2 4 5 6 7

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

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

iunique_justseen	<i>Iterator that extracts the just-seen unique elements from an iterable object</i>
------------------	---

Description

Constructs an iterator that extracts each unique element in turn from an iterable object. Order of the elements is maintained. Only the element just seen is remembered for determining uniqueness.

Usage

```
iunique_justseen(object)
```

Arguments

object an iterable object

Value

an iterator that returns the just-seen unique elements from object

Examples

```

it <- ichain(rep(1,4), rep(2, 5), 4:7, 2)
it_iunique <- iunique_justseen(it)
as.list(it_iunique) # 1 2 4 5 6 7 2

it2 <- iterators::iter(c('a', 'a', "A", 'a', 'a', "V"))
it2_iunique <- iunique_justseen(it2)
as.list(it2_iunique) # a A a V

```

izip

Iterator that iterates through several iterables concurrently.

Description

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

Usage

```
izip(...)
```

Arguments

... multiple arguments to iterate through in sequence

Value

iterator that iterates through each argument in sequence

Examples

```

it <- izip(x=1:3, y=4:6, z=7:9)
iterators::nextElem(it) # list(x=1, y=4, z=7)
iterators::nextElem(it) # list(x=2, y=5, z=8)
iterators::nextElem(it) # list(x=3, y=6, z=9)

# Sums the zip'd elements. 1 + 4 + 7, and so on.
it2 <- izip(1:3, 4:6, 7:9)
sum_zip <- sapply(it2, function(x) sum(unlist(x)))
sum_zip == c(12, 15, 18)

it3 <- izip(a=1:3, b=4:42, class=levels(iris$Species))
iterators::nextElem(it3) # list(a=1, b=4, class="setosa")
iterators::nextElem(it3) # list(a=2, b=5, class="versicolor")
iterators::nextElem(it3) # list(a=3, b=6, class="virginica")

```

izip_longest	<i>Iterator that iterates through several iterables concurrently.</i>
--------------	---

Description

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

Usage

```
izip_longest(..., fill = NA)
```

Arguments

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

Details

Although similar to [izip](#), missing values are replaced with fill if the iterables are of uneven length, and Iteration continues until the longest iterable is exhausted.

Value

iterator that iterates through each argument in sequence

Examples

```
it <- izip_longest(x=1:3, y=4:6, z=7:9)
iterators::nextElem(it) # list(x=1, y=4, z=7)
iterators::nextElem(it) # list(x=2, y=5, z=8)
iterators::nextElem(it) # list(x=3, y=6, z=9)

it2 <- izip_longest(1:3, 4:8)
iterators::nextElem(it2) # list(1, 4)
iterators::nextElem(it2) # list(2, 5)
iterators::nextElem(it2) # list(3, 6)
iterators::nextElem(it2) # list(NA, 7)
iterators::nextElem(it2) # list(NA, 8)

it3 <- izip_longest(1:2, 4:7, levels(iris$Species), fill="w00t")
iterators::nextElem(it3) # list(1, 4, "setosa")
iterators::nextElem(it3) # list(2, 5, "versicolor")
iterators::nextElem(it3) # list("w00t", 6, "virginica")
iterators::nextElem(it3) # list("w00t", 7, "w00t")
```

nth	<i>Returns the nth item of an iterator</i>
-----	--

Description

Returns the nth item of an iterator after advancing the iterator n steps ahead. If the iterator is entirely consumed, the default value is returned instead. That is, if either `n > length(iterator)` or n is 0, then the iterator is consumed.

Usage

```
nth(iterator, n, default = NA)
```

Arguments

iterator	an iterator object
n	The location of the desired element to return
default	The value to return if iterable is consumed, default is NA

Value

The nth element of the iterable or the default value

Examples

```
it <- iterators::iter(1:10)
# Returns 5
nth(it, 5)

it2 <- iterators::iter(letters)
# Returns 'e'
nth(it2, 5)

it3 <- iterators::iter(letters)
# Returns default value of NA
nth(it3, 42)

it4 <- iterators::iter(letters)
# Returns default value of "foo"
nth(it4, 42, default="foo")
```

quantify	<i>Count the number of times an iterable object is TRUE</i>
----------	---

Description

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

Usage

```
quantify(object)
```

Arguments

object an iterable object

Value

the number of TRUE elements

Examples

```
it <- iterators::iter(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)
```

stop_iteration	<i>Helper function that determines whether an object inherits from a StopIteration exception</i>
----------------	--

Description

Returns TRUE if the object resulted from a StopIteration exception when `nextElem` is called, and FALSE otherwise.

Usage

```
stop_iteration(object)
```

Arguments

object an R object

Value

TRUE if object resulted from a StopIteration exception. Otherwise, FALSE.

take	<i>Return the first n elements of an iterable object as a list</i>
------	--

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

```
take(object, n = 1)
```

Arguments

object	an iterable object
n	the number of elements to return in the list

Value

a list of the first n items of the iterable object

Examples

```
take(iterators::iter(1:10), 3) # 1 2 3
take(iterators::iter(1:5), 10) # 1 2 3 4 5
```

try_nextElem	<i>Calls iterators::nextElem(). If error, returns default value.</i>
--------------	--

Description

Returns the next element of object. In the case a StopIteration exception is thrown, the default value is returned instead.

Usage

```
try_nextElem(object, default = NA, silent = TRUE)
```

Arguments

object	an iterable object
default	default value returned if a StopIteration exception is thrown
silent	Should any errors be suppressed without explicitly notifying the user? Default. Yes

Value

the next element of object

Index

consume, [2](#)

dotproduct, [3](#)

expand.grid, [15](#)

ichain, [4](#)

ichunk, [4](#)

icompress, [5](#)

icount, [6](#)

icycle, [7](#)

idropwhile, [8](#)

ienum (ienumerate), [9](#)

ienumerate, [9](#)

ifilter, [10](#)

ifilterfalse (ifilter), [10](#)

ilength, [11](#)

imap, [6](#), [12](#), [12](#), [16](#), [20](#)

ipad, [13](#)

ipairwise, [14](#)

iproduct, [14](#)

irep, [15](#)

irep_len (irep), [15](#)

irepeat, [16](#)

iroundrobin, [17](#)

is_iterator, [21](#)

iseq, [18](#)

iseq_along (iseq), [18](#)

iseq_len (iseq), [18](#)

islice, [19](#)

istar (istarmap), [20](#)

istarmap, [20](#)

itabulate, [22](#)

itakewhile, [22](#)

itee, [23](#)

iter, [23](#)

iter_deepcopy, [24](#)

iter_length, [25](#)

itertools2, [24](#)

itertools2-package (itertools2), [24](#)

iunique, [25](#)

iunique_justseen, [26](#)

izip, [16](#), [27](#), [28](#)

izip_longest, [28](#)

Map, [12](#), [20](#)

nextElem, [9](#), [30](#)

nth, [29](#)

package-itertools2 (itertools2), [24](#)

quantify, [30](#)

rep, [16](#)

seq, [18](#)

sort, [25](#)

stop_iteration, [30](#)

take, [31](#)

try_nextElem, [31](#)