

Package ‘peruse’

March 8, 2021

Title A Tidy API for Sequence Iteration and Set Comprehension

Version 0.3.1

Description A friendly API for sequence iteration and set comprehension.

License GPL-2

URL <https://github.com/jacgoldsm/peruse>,
<https://jacgoldsm.github.io/peruse/>

BugReports <https://github.com/jacgoldsm/peruse/issues>

Encoding UTF-8

Imports rlang

Depends magrittr

LazyData true

RoxygenNote 7.1.1

Suggests testthat, purrr

NeedsCompilation no

Author Jacob Goldsmith [aut, cre]

Maintainer Jacob Goldsmith <jacobg314@hotmail.com>

Repository CRAN

Date/Publication 2021-03-08 07:20:02 UTC

R topics documented:

clone	2
current	2
is_Iterator	3
Iterator	3
move	4
range	5
sets	6
yield	8
yield_while	9

Index**10**

clone	<i>clone</i>
-------	--------------

Description

Clone an Iterator, making an exact copy that can then be modified separately. This is a simple wrapper around `rlang::env_clone()`. Optionally, override old initial parameters.

Usage

```
clone(iter, ...)
```

Arguments

<code>iter</code>	an Iterator object
<code>...</code>	optionally override the <code>\$initial</code> parameters in <code>iter</code>

Value

a copy of the Iterator passed as a parameter

Examples

```
it <- Iterator({m <- m + n}, list(m = 0, n = 1), m)
other <- clone(it)
yield_next(it)
current(other) == current(it) # false

it2 <- clone(other, n = 5)
yield_next(it2)
it2$initial$n # 5
```

current	<i>Get the current value of an Iterator without changing its state</i>
---------	--

Description

An Iterator yields a variable every time `yield_next()` is called. Get the current value of that variable without changing the state of the Iterator.

Usage

```
current(iter)
```

Arguments

iter An Iterator object

Value

The current value of iter

is_Iterator *Test if an object is an Iterator*

Description

Test if an object is an Iterator

Usage

```
is_Iterator(list)
```

Arguments

list Object to test

Iterator *Making an Irregular Sequence Iterator*

Description

Create an Iterator object, where the user defines a sequence and a set of initial values, and then calls `yield_next()` to generate the next element of the sequence. Iterators are R environments, which means they are modified in place, even when passed as arguments to functions. To make a copy of an Iterator that can be modified separately, see [clone\(\)](#).

Usage

```
Iterator(result, initial, yield)
```

Arguments

result R expression to run each time 'yield_next' is called
initial named list or vector; declare and initialize every variable that appears in 'result'
yield variable to yield when 'yield_next()' is called

Value

An environment object of S3 type Iterator

Note

The expression to be evaluated can include constant values not defined in `$initial` as long as they are defined in the enclosure *of where `yield_next()` is called*, not where the Iterator is created. These values will not vary from iteration to iteration (unless you do something strange in the code, like including `<<-` in `$result`.)

See Also

[yield_next\(\)](#), [yield_while\(\)](#), [current\(\)](#) [rlang::qq_show\(\)](#)

Examples

```
#Create the Collatz sequence starting with 50 and print out the first 30 elements
collatz <- Iterator({
  if (n %% 2 == 0) n <- n / 2 else n <- n*3 + 1
},
  initial = c(n = 50),
  yield = n)

seq <- yield_more(collatz, 30)

# If you want to define the expression outside the Iterator, use [quote()] and `!!`:
expr <- quote(if (n %% 2 == 0) n <- n / 2 else n <- n*3 + 1)
collatz <- Iterator(!!expr,
  c(n = 50),
  n)

# using objects defined outside `initial`:
# Note that `n` in `initial` overrides the global `n`
m <- 100
n <- 10
it <- Iterator({out <- n + m},
  initial = c(n = -10),
  yield = out)

yield_next(it)

# environments are modified in place, so be aware:
it <- Iterator({m <- m + 1}, c(m = 0), m)
other <- it
yield_next(it)
current(other)
```

Description

Increments the Iterator without returning anything. `move_more()` repeats `move_next()` a specified number of times. `move_while()` repeats `move_next()` until a condition is met. Refer to the number of the current iteration with `.iter`.

Usage

```
move_next(iter)

move_more(iter, more = 1L)

move_while(iter, cond)
```

Arguments

<code>iter</code>	An Iterator object object
<code>more</code>	How many times to iterate
<code>cond</code>	A quoted logical expression involving some variable(s) in <code>iter\$initial</code> , so that <code>move_next()</code> continues being called while the expression returns TRUE

Examples

```
primes <- 2:10000 %>%
  that_for_all(range(2, .x)) %>%
  we_have(~.x %% .y != 0, "Iterator")
current(primes)
move_more(primes, 100)
current(primes)
```

range

Python-style range function

Description

Wrapper around `base::seq()` that replaces the maximal end value with the supremum and returns an empty vector if `b <= a`, in the style of Python's `range()`. Note that `peruse::range` views end as a supremum, not a maximum, thus `range(a,b)` is equivalent to the set `[a,b)` when `a < b` or `{}` when `b >= a`.

Usage

```
range(a, b, ...)
```

Arguments

a	minimum
b	supremum
...	other params passed to <code>base::seq()</code>

See Also

[base::seq\(\)](#)

Examples

```
range(1,5)
range(9,10)
range(1,6, by = 2)
```

sets

R Set Comprehension

Description

Set comprehension with the magrittr Pipe. Always use the basic syntax:

`.x %>% that_for_all(.y) %>% we_have_*(f(.x, .y))`, but see the examples for more detail.

Usage

```
that_for_all(.x, .y)
```

```
that_for_any(.x, .y)
```

```
we_have(that_for, formula, result = "vector")
```

Arguments

.x	A set, represented as either an atomic vector or a list
.y	A set to compare to .x
that_for	A list passed to <code>we_have()</code> —can be ignored with proper syntax
formula	A function, lambda, or formula. Must be understood by <code>rlang::as_function()</code>
result	Should the expression return a vector or an Iterator?

Details

formula can be anything that is recognized as a function by `rlang::as_function()`. See the examples for how to specify the end of a sequence when used with an Iterator.

Handling missing values in these expressions is possible and sometimes desirable but potentially painful because NA values can't be compared with normal operators. See the README for a detailed example.

Note that `.x %>% that_for_all(.y)` is vacuously true if `.y` is empty, while `.x %>% that_for_any(.y)` is vacuously false if `.y` is empty.

Value

For `that_for_all()` and `that_for_any()`, an object of S3 class `that_for_all` or `that_for_any`. For `we_have()`, a vector of the same type as `.x` if `return == 'vector'` and an Iterator object if `return == 'Iterator'`.

Note

if `.y` is an numeric vector, you probably want a value obtained from `range(start, end)` rather than `start:end` or `seq.int(start, end)`, as when `start` is greater than `end` you want an empty vector rather than counting backwards. Note that `range()` views `end` as a supremum, not a maximum, thus `range(a, b)` is equivalent to the set `[a, b)` when `a < b` or the empty set when `b >= a`.

Also note that there is some indirection in the way that `.x` and `.y` are referenced in the formula. In the function `we_have()`, the actual name of the two sets is `.x` and `.y`. That is what makes the function interface work, e.g. `function(.x, .y) .x - .y`. On the other hand, `purrr`-style lambda expressions, e.g. `~.x - .y`, use positional arguments, where `.x` is the first argument and `.y` is the second argument, no matter their names. Because those are actually their names, this difference should never matter.

See Also

The implementation of these functions involves code adapted from `purrr::every()` and `purrr::some()`, by Lionel Henry, Hadley Wickham, and RStudio, available under the MIT license.

Examples

```
2:100 %>% that_for_all(range(2, .x)) %>% we_have(function(.x, .y) .x %% .y != 0) #is the same as
2:100 %>% that_for_all(range(2, .x)) %>% we_have(~.x %% .y) # 0 = F, (not 0) = T
#c.f.
primes <- 2:100 %>% that_for_all(range(2, .x)) %>% we_have(~.x %% .y, "Iterator")
yield_next(primes)
primes2 <- clone(primes)

# Refer to the vector .x with `x_vector` and the current index of that vector with `i`
# For example, to yield to the end of the sequence:
yield_while(primes, x_vector[i] <= length(x_vector))
# `finished` is an alias for `x_vector[i] > length(x_vector)`
# Equivalent to previous expression:
yield_while(primes2, !finished)
{c("I", "Don't", "wan't", "chicken")} %>%
```

```

        that_for_all("`") %>%
        we_have(~grepl(.y, .x))}
#Twin primes 1 through 100
primes <- 2:100 %>% that_for_all(range(2, .x)) %>% we_have(~.x %% .y)
primes %>% that_for_any(primes) %>% we_have(~abs(.x - .y) == 2)
#Prime numbers 1 through 100 that are two away from a square number
(2:100 %>% that_for_all(range(2, .x)) %>% we_have(~.x %% .y)) %>%
  that_for_any(range(2, .x)) %>% we_have(~sqrt(.x + 2) == .y | sqrt(.x - 2) == .y)

```

yield

Increment an Iterator and Return the Next Value(s)

Description

Finds the value of the next iteration(s) of an Iterator object and increments the Iterator to the next value(s). `yield_more()` repeats `yield_next()` a specified number of times. Refer to the number of the current iteration in `yield_more()` with `.iter`.

Usage

```
yield_next(iter)
```

```
yield_more(iter, more = 1L)
```

Arguments

<code>iter</code>	An Iterator object
<code>more</code>	How many values to yield

Value

An object of whatever type `result` evaluates to from the Iterator, or a vector of that type in the case of `yield_more(iter, more > 1L)`.

Examples

```

primes <- 2:10000 %>%
  that_for_all(range(2, .x)) %>%
  we_have(~.x %% .y != 0, "Iterator")

sequence <- yield_more(primes, 100)

# use `.iter` to reference the current iteration
rwd <- Iterator({
  set.seed(seeds[.iter])
  n <- n + sample(c(-1L, 1L), size = 1L, prob = c(0.25, 0.75))
},
  initial = list(n = 0, seeds = 1:100),

```



```

        yield = n)
yield_more(rwd, 100)

```

```

yield_while      yield_while

```

Description

Keep yielding the next element of an Iterator while a condition is met. A condition is a logical expression involving variables in `iter$initial` or variables that are defined in the enclosure. Refer to the number of the current iteration with `.iter`.

Usage

```
yield_while(iter, cond)
```

Arguments

<code>iter</code>	An Iterator object
<code>cond</code>	A logical expression involving some variable(s) in <code>iter\$initial</code> or in the enclosure, so that <code>yield_next()</code> continues being called while the expression returns TRUE

Examples

```

collatz <- Iterator({
  if (n %% 2 == 0) n <- n / 2 else n <- n*3 + 1
},
  initial = list(n = 50),
  yield = n)
yield_while(collatz, n != 1L)

p_success <- 0.5
threshold <- 100
seeds <- 1000:1e6
iter <- Iterator({
  set.seed(seeds[.iter])
  n <- n + sample(c(1,-1), 1, prob = c(p_success, 1 - p_success))
},
  list(n = 0),
  n)
sequence <- yield_while(iter, n <= threshold)

```

Index

`<<-`, 4

`base::seq()`, 6

`clone`, 2

`clone()`, 3

`current`, 2

`current()`, 4

`is_Iterator`, 3

`Iterator`, 3

`move`, 4

`move_more (move)`, 4

`move_next (move)`, 4

`move_while (move)`, 4

`purrr::every()`, 7

`purrr::some()`, 7

`range`, 5

`range()`, 7

`rlang::as_function()`, 6, 7

`rlang::env_clone()`, 2

`rlang::qq_show()`, 4

`sets`, 6

`that_for_all (sets)`, 6

`that_for_any (sets)`, 6

`we_have (sets)`, 6

`we_have()`, 6

`yield`, 8

`yield_more (yield)`, 8

`yield_next (yield)`, 8

`yield_next()`, 4

`yield_while`, 9

`yield_while()`, 4