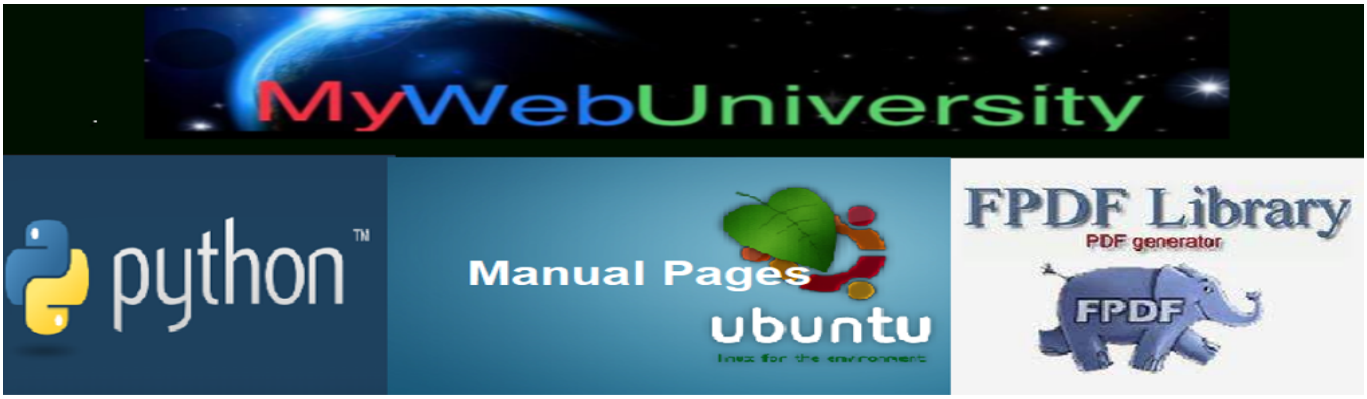

Rocky Enterprise Linux 9.2 Manual Pages on command 'List::MoreUtils.3pm'

$ man List::MoreUtils.3pm

List::MoreUtils(3pm) User Contributed Perl Documentation List::MoreUtils(3pm)

NAME

List::MoreUtils - Provide the stuff missing in List::Util

SYNOPSIS

```
# import specific functions
use List::MoreUtils qw(any uniq);
if ( any { /foo/ } uniq @has_duplicates ) {
    # do stuff
}
# import everything
use List::MoreUtils ':all';
# import by API
# has "original" any/all/none/notall behavior
use List::MoreUtils ':like_0.22';
# 0.22 + bsearch
use List::MoreUtils ':like_0.24';
# has "simplified" any/all/none/notall behavior + (n)sort_by
use List::MoreUtils ':like_0.33';
```

DESCRIPTION

List::MoreUtils provides some trivial but commonly needed functionality on lists which is not going to go into List::Util.

All of the below functions are implementable in only a couple of lines of Perl code. Using the functions from this module however should give slightly better performance as

everything is implemented in C. The pure-Perl implementation of these functions only serves as a fallback in case the C portions of this module couldn't be compiled on this machine.

EXPORTS

Default behavior

Nothing by default. To import all of this module's symbols use the ":all" tag. Otherwise functions can be imported by name as usual:

```
use List::MoreUtils ':all';
use List::MoreUtils qw{ any firstidx };
```

Because historical changes to the API might make upgrading List::MoreUtils difficult for some projects, the legacy API is available via special import tags.

Like version 0.22 (last release with original API)

This API was available from 2006 to 2009, returning undef for empty lists on "all"/"any"/"none"/"notall":

```
use List::MoreUtils ':like_0.22';
```

This import tag will import all functions available as of version 0.22. However, it will import "any_u" as "any", "all_u" as "all", "none_u" as "none", and "notall_u" as "notall".

Like version 0.24 (first incompatible change)

This API was available from 2010 to 2011. It changed the return value of "none" and added the "bsearch" function.

```
use List::MoreUtils ':like_0.24';
```

This import tag will import all functions available as of version 0.24. However it will import "any_u" as "any", "all_u" as "all", and "notall_u" as "notall". It will import "none" as described in the documentation below (true for empty list).

Like version 0.33 (second incompatible change)

This API was available from 2011 to 2014. It is widely used in several CPAN modules and thus it's closest to the current API. It changed the return values of "any", "all", and "notall". It added the "sort_by" and "nsort_by" functions and the "distinct" alias for "uniq". It omitted "bsearch".

```
use List::MoreUtils ':like_0.33';
```

This import tag will import all functions available as of version 0.33. Note: it will not import "bsearch" for consistency with the 0.33 API.

FUNCTIONS

Junctions

Treatment of an empty list

There are two schools of thought for how to evaluate a junction on an empty list:

? Reduction to an identity (boolean)

? Result is undefined (three-valued)

In the first case, the result of the junction applied to the empty list is determined by a mathematical reduction to an identity depending on whether the underlying comparison is "or" or "and". Conceptually:

	"any are true"	"all are true"
2 elements:	A || B || 0	A && B && 1
1 element:	A || 0	A && 1
0 elements:	0	1

In the second case, three-value logic is desired, in which a junction applied to an empty list returns "undef" rather than true or false

Junctions with a "_u" suffix implement three-valued logic. Those without are boolean.

all BLOCK LIST

all_u BLOCK LIST

Returns a true value if all items in LIST meet the criterion given through BLOCK. Sets $_ for each item in LIST in turn:

```
print "All values are non-negative"
  if all { $_ >= 0 } ($x, $y, $z);
```

For an empty LIST, "all" returns true (i.e. no values failed the condition) and "all_u" returns "undef".

Thus, "all_u(@list)" is equivalent to "@list ? all(@list) : undef".

Note: because Perl treats "undef" as false, you must check the return value of "all_u" with "defined" or you will get the opposite result of what you expect.

any BLOCK LIST

any_u BLOCK LIST

Returns a true value if any item in LIST meets the criterion given through BLOCK. Sets $_ for each item in LIST in turn:

```
print "At least one non-negative value"
  if any { $_ >= 0 } ($x, $y, $z);
```

For an empty LIST, "any" returns false and "any_u" returns "undef".

Thus, "any_u(@list)" is equivalent to "@list ? any(@list) : undef".

none BLOCK LIST

none_u BLOCK LIST

Logically the negation of "any". Returns a true value if no item in LIST meets the criterion given through BLOCK. Sets $_ for each item in LIST in turn:

```
print "No non-negative values"
  if none { $_ >= 0 } ($x, $y, $z);
```

For an empty LIST, "none" returns true (i.e. no values failed the condition) and "none_u" returns "undef".

Thus, "none_u(@list)" is equivalent to "@list ? none(@list) : undef".

Note: because Perl treats "undef" as false, you must check the return value of "none_u" with "defined" or you will get the opposite result of what you expect.

notall BLOCK LIST

notall_u BLOCK LIST

Logically the negation of "all". Returns a true value if not all items in LIST meet the criterion given through BLOCK. Sets $_ for each item in LIST in turn:

```
print "Not all values are non-negative"
  if notall { $_ >= 0 } ($x, $y, $z);
```

For an empty LIST, "notall" returns false and "notall_u" returns "undef".

Thus, "notall_u(@list)" is equivalent to "@list ? notall(@list) : undef".

one BLOCK LIST

one_u BLOCK LIST

Returns a true value if precisely one item in LIST meets the criterion given through BLOCK. Sets $_ for each item in LIST in turn:

```
print "Precisely one value defined"
    if one { defined($_) } @list;
```

Returns false otherwise.

For an empty LIST, "one" returns false and "one_u" returns "undef".

The expression "one BLOCK LIST" is almost equivalent to "1 == true BLOCK LIST", except for short-cutting. Evaluation of BLOCK will immediately stop at the second true value.

Transformation

apply BLOCK LIST

Applies BLOCK to each item in LIST and returns a list of the values after BLOCK has been applied. In scalar context, the last element is returned. This function is similar to "map" but will not modify the elements of the input list:

```
  my @list = (1 .. 4);
  my @mult = apply { $_ *= 2 } @list;
  print "\@list = @list\n";
  print "\@mult = @mult\n";
  __END__
  @list = 1 2 3 4
  @mult = 2 4 6 8
```

Think of it as syntactic sugar for

```
  for (my @mult = @list) { $_ *= 2 }
```

insert_after BLOCK VALUE LIST

Inserts VALUE after the first item in LIST for which the criterion in BLOCK is true. Sets $_ for each item in LIST in turn.

```
  my @list = qw/This is a list/;
  insert_after { $_ eq "a" } "longer" => @list;
  print "@list";
  __END__
  This is a longer list
```

insert_after_string STRING VALUE LIST

Inserts VALUE after the first item in LIST which is equal to STRING.

```
  my @list = qw/This is a list/;
  insert_after_string "a", "longer" => @list;
  print "@list";
  __END__
  This is a longer list
```

pairwise BLOCK ARRAY1 ARRAY2

Evaluates BLOCK for each pair of elements in ARRAY1 and ARRAY2 and returns a new list consisting of BLOCK's return values. The two elements are set to $a and $b. Note that those two are aliases to the original value so changing them will modify the input arrays.

```
  @a = (1 .. 5);
  @b = (11 .. 15);
```

```
  @x = pairwise { $a + $b } @a, @b;     # returns 12, 14, 16, 18, 20
  # mesh with pairwise
  @a = qw/a b c/;
  @b = qw/1 2 3/;
  @x = pairwise { ($a, $b) } @a, @b;    # returns a, 1, b, 2, c, 3
```

mesh ARRAY1 ARRAY2 [ARRAY3 ...]

zip ARRAY1 ARRAY2 [ARRAY3 ...]

Returns a list consisting of the first elements of each array, then the second, then the third, etc, until all arrays are exhausted.

Examples:

```
  @x = qw/a b c d/;
  @y = qw/1 2 3 4/;
  @z = mesh @x, @y;         # returns a, 1, b, 2, c, 3, d, 4
  @a = ('x');
  @b = ('1', '2');
  @c = qw/zip zap zot/;
  @d = mesh @a, @b, @c;   # x, 1, zip, undef, 2, zap, undef, undef, zot
```

"zip" is an alias for "mesh".

zip6

zip_unflatten

Returns a list of arrays consisting of the first elements of each array, then the second, then the third, etc, until all arrays are exhausted.

```
  @x = qw/a b c d/;
  @y = qw/1 2 3 4/;
  @z = zip6 @x, @y;         # returns [a, 1], [b, 2], [c, 3], [d, 4]
  @a = ('x');
  @b = ('1', '2');
  @c = qw/zip zap zot/;
  @d = zip6 @a, @b, @c;     # [x, 1, zip], [undef, 2, zap], [undef, undef, zot]
```

"zip_unflatten" is an alias for "zip6".

listcmp ARRAY0 ARRAY1 [ARRAY2 ...]

Returns an associative list of elements and every id of the list it was found in. Allows easy implementation of @a & @b, @a | @b, @a ^ @b and so on. Undefined entries in any

given array are skipped.

```
  my @a = qw(one two three four five six seven eight nine ten eleven twelve thirteen);
  my @b = qw(two three five seven eleven thirteen seventeen);
  my @c = qw(one one two three five eight thirteen twentyone);
  my %cmp = listcmp @a, @b, @c; # returns (one => [0, 2], two => [0, 1, 2], three => [0, 1, 2], four => [0], ...)
  my @seq = (1, 2, 3);
  my @prim = (undef, 2, 3, 5);
  my @fib = (1, 1, 2);
  my %cmp = listcmp @seq, @prim, @fib;
  # returns ( 1 => [0, 2], 2 => [0, 1, 2], 3 => [0, 1], 5 => [1] )
```

arrayify LIST[,LIST[,LIST...]]

Returns a list consisting of each element of given arrays. Recursive arrays are flattened, too.

```
  @a = (1, [[2], 3], 4, [5], 6, [7], 8, 9);
  @l = arrayify @a;         # returns 1, 2, 3, 4, 5, 6, 7, 8, 9
```

uniq LIST

distinct LIST

Returns a new list by stripping duplicate values in LIST by comparing the values as hash keys, except that undef is considered separate from ''. The order of elements in the returned list is the same as in LIST. In scalar context, returns the number of unique elements in LIST.

```
  my @x = uniq 1, 1, 2, 2, 3, 5, 3, 4; # returns 1 2 3 5 4
  my $x = uniq 1, 1, 2, 2, 3, 5, 3, 4; # returns 5
  # returns "Mike", "Michael", "Richard", "Rick"
  my @n = distinct "Mike", "Michael", "Richard", "Rick", "Michael", "Rick"
  # returns "A8", "", undef, "A5", "S1"
  my @s = distinct "A8", "", undef, "A5", "S1", "A5", "A8"
  # returns "Giulia", "Giulietta", undef, "", 156, "GTA", "GTV", 159, "Brera", "4C"
  my @w = uniq "Giulia", "Giulietta", undef, "", 156, "GTA", "GTV", 159, "Brera", "4C", "Giulietta", "Giulia"
```

"distinct" is an alias for "uniq".

RT#49800 can be used to give feedback about this behavior.

singleton LIST

Returns a new list by stripping values in LIST occurring more than once by comparing the

values as hash keys, except that undef is considered separate from ''. The order of elements in the returned list is the same as in LIST. In scalar context, returns the number of elements occurring only once in LIST.

```
my @x = singleton 1,1,2,2,3,4,5 # returns 3 4 5
```

duplicates LIST

Returns a new list by stripping values in LIST occurring less than twice by comparing the values as hash keys, except that undef is considered separate from ''. The order of elements in the returned list is the same as in LIST. In scalar context, returns the number of elements occurring more than once in LIST.

```
my @y = duplicates 1,1,2,4,7,2,3,4,6,9; #returns 1,2,4
```

frequency LIST

Returns an associative list of distinct values and the corresponding frequency.

```
my @f = frequency values %radio_nrw; # returns (
# 'Deutschlandfunk (DLF)' => 9, 'WDR 3' => 10,
# 'WDR 4' => 11, 'WDR 5' => 14, 'WDR Eins Live' => 14,
# 'Deutschlandradio Kultur' => 8,...)
```

occurrences LIST

Returns a new list of frequencies and the corresponding values from LIST.

```
my @o = occurrences ((1) x 3, (2) x 4, (3) x 2, (4) x 7, (5) x 2, (6) x 4);
# @o = (undef, undef, [3, 5], [1], [2, 6], undef, undef, [4]);
```

mode LIST

Returns the modal value of LIST. In scalar context, just the modal value is returned, in list context all probes occurring modal times are returned, too.

```
my @m = mode ((1) x 3, (2) x 4, (3) x 2, (4) x 7, (5) x 2, (6) x 4, (7) x 3, (8) x 7);
# @m = (7, 4, 8) - bimodal LIST
```

slide BLOCK LIST

The function "slide" operates on pairs of list elements like:

```
my @s = slide { "$a and $b" } (0..3);
# @s = ("0 and 1", "1 and 2", "2 and 3")
```

The idea behind this function is a kind of magnifying glass that is moved along a list and calls "BLOCK" every time the next list item is reached.

Partitioning

after BLOCK LIST

Returns a list of the values of LIST after (and not including) the point where BLOCK returns a true value. Sets $_ for each element in LIST in turn.

```
  @x = after { $_ % 5 == 0 } (1..9);    # returns 6, 7, 8, 9
```

after_incl BLOCK LIST

Same as "after" but also includes the element for which BLOCK is true.

before BLOCK LIST

Returns a list of values of LIST up to (and not including) the point where BLOCK returns a true value. Sets $_ for each element in LIST in turn.

before_incl BLOCK LIST

Same as "before" but also includes the element for which BLOCK is true.

part BLOCK LIST

Partitions LIST based on the return value of BLOCK which denotes into which partition the current value is put.

Returns a list of the partitions thusly created. Each partition created is a reference to an array.

```
  my $i = 0;
  my @part = part { $i++ % 2 } 1 .. 8;   # returns [1, 3, 5, 7], [2, 4, 6, 8]
```

You can have a sparse list of partitions as well where non-set partitions will be undef:

```
  my @part = part { 2 } 1 .. 10;            # returns undef, undef, [ 1 .. 10 ]
```

Be careful with negative values, though:

```
  my @part = part { -1 } 1 .. 10;
  __END__
  Modification of non-creatable array value attempted, subscript -1 ...
```

Negative values are only ok when they refer to a partition previously created:

```
  my @idx  = ( 0, 1, -1 );
  my $i    = 0;
  my @part = part { $idx[$i++ % 3] } 1 .. 8; # [1, 4, 7], [2, 3, 5, 6, 8]
```

samples COUNT LIST

Returns a new list containing COUNT random samples from LIST. Is similar to "shuffle" in List::Util, but stops after COUNT.

```
  @r  = samples 10, 1..10; # same as shuffle
  @r2 = samples 5, 1..10; # gives 5 values from 1..10;
```

each_array ARRAY1 ARRAY2 ...

Creates an array iterator to return the elements of the list of arrays ARRAY1, ARRAY2 throughout ARRAYn in turn. That is, the first time it is called, it returns the first element of each array. The next time, it returns the second elements. And so on, until all elements are exhausted.

This is useful for looping over more than one array at once:

```
my $ea = each_array(@a, @b, @c);
while ( my ($a, $b, $c) = $ea->() )   { .... }
```

The iterator returns the empty list when it reached the end of all arrays.

If the iterator is passed an argument of '"index"', then it returns the index of the last fetched set of values, as a scalar.

each_arrayref LIST

Like each_array, but the arguments are references to arrays, not the plain arrays.

natatime EXPR, LIST

Creates an array iterator, for looping over an array in chunks of $n items at a time. (n at a time, get it?). An example is probably a better explanation than I could give in words.

Example:

```
my @x = ('a' .. 'g');
my $it = natatime 3, @x;
while (my @vals = $it->())
{
  print "@vals\n";
}
```

This prints

```
a b c
d e f
g
```

slideatatime STEP, WINDOW, LIST

Creates an array iterator, for looping over an array in chunks of "$windows-size" items at a time.

The idea behind this function is a kind of magnifying glass (finer controllable compared to "slide") that is moved along a list.

Example:

```
my @x = ('a' .. 'g');
my $it = slideatatime 2, 3, @x;
while (my @vals = $it->())
{
  print "@vals\n";
}
```

This prints

```
a b c
c d e
e f g
g
```

Searching

firstval BLOCK LIST

first_value BLOCK LIST

Returns the first element in LIST for which BLOCK evaluates to true. Each element of LIST is set to $_ in turn. Returns "undef" if no such element has been found.

"first_value" is an alias for "firstval".

onlyval BLOCK LIST

only_value BLOCK LIST

Returns the only element in LIST for which BLOCK evaluates to true. Sets $_ for each item in LIST in turn. Returns "undef" if no such element has been found.

"only_value" is an alias for "onlyval".

lastval BLOCK LIST

last_value BLOCK LIST

Returns the last value in LIST for which BLOCK evaluates to true. Each element of LIST is set to $_ in turn. Returns "undef" if no such element has been found.

"last_value" is an alias for "lastval".

firstres BLOCK LIST

first_result BLOCK LIST

Returns the result of BLOCK for the first element in LIST for which BLOCK evaluates to true. Each element of LIST is set to $_ in turn. Returns "undef" if no such element has been found.

"first_result" is an alias for "firstres".

onlyres BLOCK LIST

only_result BLOCK LIST

Returns the result of BLOCK for the first element in LIST for which BLOCK evaluates to true. Sets $_ for each item in LIST in turn. Returns "undef" if no such element has been found.

"only_result" is an alias for "onlyres".

lastres BLOCK LIST

last_result BLOCK LIST

Returns the result of BLOCK for the last element in LIST for which BLOCK evaluates to true. Each element of LIST is set to $_ in turn. Returns "undef" if no such element has been found.

"last_result" is an alias for "lastres".

indexes BLOCK LIST

Evaluates BLOCK for each element in LIST (assigned to $_) and returns a list of the indices of those elements for which BLOCK returned a true value. This is just like "grep" only that it returns indices instead of values:

```
  @x = indexes { $_ % 2 == 0 } (1..10);   # returns 1, 3, 5, 7, 9
```

firstidx BLOCK LIST

first_index BLOCK LIST

Returns the index of the first element in LIST for which the criterion in BLOCK is true. Sets $_ for each item in LIST in turn:

```
  my @list = (1, 4, 3, 2, 4, 6);
  printf "item with index %i in list is 4", firstidx { $_ == 4 } @list;
  __END__
  item with index 1 in list is 4
```

Returns "-1" if no such item could be found.

"first_index" is an alias for "firstidx".

onlyidx BLOCK LIST

only_index BLOCK LIST

Returns the index of the only element in LIST for which the criterion in BLOCK is true. Sets $_ for each item in LIST in turn:

```
    my @list = (1, 3, 4, 3, 2, 4);
```

```
  printf "uniqe index of item 2 in list is %i", onlyidx { $_ == 2 } @list;
  __END__
  unique index of item 2 in list is 4
```

Returns "-1" if either no such item or more than one of these has been found.

"only_index" is an alias for "onlyidx".

lastidx BLOCK LIST

last_index BLOCK LIST

Returns the index of the last element in LIST for which the criterion in BLOCK is true.

Sets $_ for each item in LIST in turn:

```
  my @list = (1, 4, 3, 2, 4, 6);
  printf "item with index %i in list is 4", lastidx { $_ == 4 } @list;
  __END__
  item with index 4 in list is 4
```

Returns "-1" if no such item could be found.

"last_index" is an alias for "lastidx".

Sorting

sort_by BLOCK LIST

Returns the list of values sorted according to the string values returned by the KEYFUNC block or function. A typical use of this may be to sort objects according to the string value of some accessor, such as

```
  sort_by { $_->name } @people
```

The key function is called in scalar context, being passed each value in turn as both $_ and the only argument in the parameters, @_. The values are then sorted according to string comparisons on the values returned. This is equivalent to

```
  sort { $a->name cmp $b->name } @people
```

except that it guarantees the name accessor will be executed only once per value. One interesting use-case is to sort strings which may have numbers embedded in them "naturally", rather than lexically.

```
  sort_by { s/(\d+)/sprintf "%09d", $1/eg; $_ } @strings
```

This sorts strings by generating sort keys which zero-pad the embedded numbers to some level (9 digits in this case), helping to ensure the lexical sort puts them in the correct order.

nsort_by BLOCK LIST

Similar to sort_by but compares its key values numerically.

qsort BLOCK ARRAY

This sorts the given array in place using the given compare code. Except for tiny compare code like "$a <=> $b", qsort is much faster than Perl's "sort" depending on the version.

Compared 5.8 and 5.26:

```
 my @rl;
 for(my $i = 0; $i < 1E6; ++$i) { push @rl, rand(1E5) }
 my $idx;
 sub ext_cmp { $_[0] <=> $_[1] }
 cmpthese( -60, {
    'qsort' => sub {
        my @qrl = @rl;
        qsort { ext_cmp($a, $b) } @qrl;
        $idx = bsearchidx { ext_cmp($_, $rl[0]) } @qrl
    },
    'reverse qsort' => sub {
        my @qrl = @rl;
        qsort { ext_cmp($b, $a) } @qrl;
        $idx = bsearchidx { ext_cmp($rl[0], $_) } @qrl
    },
    'sort' => sub {
        my @srl = @rl;
        @srl = sort { ext_cmp($a, $b) } @srl;
        $idx = bsearchidx { ext_cmp($_, $rl[0]) } @srl
    },
    'reverse sort' => sub {
        my @srl = @rl;
        @srl = sort { ext_cmp($b, $a) } @srl;
        $idx = bsearchidx { ext_cmp($rl[0], $_) } @srl
    },
 });
```

5.8 results

```
                s/iter  reverse sort          sort reverse qsort         qsort
```

reverse sort 6.21 -- -0% -8% -10%

sort 6.19 0% -- -7% -10%

reverse qsort 5.73 8% 8% -- -2%

qsort 5.60 11% 11% 2% --

5.26 results

s/iter reverse sort sort reverse qsort qsort

reverse sort 4.54 -- -0% -96% -96%

sort 4.52 0% -- -96% -96%

reverse qsort 0.203 2139% 2131% -- -19%

qsort 0.164 2666% 2656% 24% --

Use it where external data sources might have to be compared (think of Unix::Statgrab "tables").

"qsort" is available from List::MoreUtils::XS only. It's insane to maintain a wrapper around Perl's sort nor having a pure Perl implementation. One could create a flip-book in same speed as PP runs a qsort.

Searching in sorted Lists

bsearch BLOCK LIST

Performs a binary search on LIST which must be a sorted list of values. BLOCK must return a negative value if the current element (stored in $_) is smaller, a positive value if it is bigger and zero if it matches.

Returns a boolean value in scalar context. In list context, it returns the element if it was found, otherwise the empty list.

bsearchidx BLOCK LIST

bsearch_index BLOCK LIST

Performs a binary search on LIST which must be a sorted list of values. BLOCK must return a negative value if the current element (stored in $_) is smaller, a positive value if it is bigger and zero if it matches.

Returns the index of found element, otherwise "-1".

"bsearch_index" is an alias for "bsearchidx".

lower_bound BLOCK LIST

Returns the index of the first element in LIST which does not compare less than val.

Technically it's the first element in LIST which does not return a value below zero when passed to BLOCK.

```
  @ids = (1, 1, 2, 2, 3, 3, 3, 3, 3, 3, 4, 4, 4, 4, 5, 5, 6, 7, 7, 7, 8, 8, 9, 9, 9, 9, 9, 11, 13, 13, 13, 17);
  $lb = lower_bound { $_ <=> 2 } @ids; # returns 2
  $lb = lower_bound { $_ <=> 4 } @ids; # returns 10
```

lower_bound has a complexity of O(log n).

upper_bound BLOCK LIST

Returns the index of the first element in LIST which does not compare greater than val.

Technically it's the first element in LIST which does not return a value below or equal to

zero when passed to BLOCK.

```
  @ids = (1, 1, 2, 2, 3, 3, 3, 3, 3, 3, 4, 4, 4, 4, 5, 5, 6, 7, 7, 7, 8, 8, 9, 9, 9, 9, 9, 11, 13, 13, 13, 17);
  $lb = upper_bound { $_ <=> 2 } @ids; # returns 4
  $lb = upper_bound { $_ <=> 4 } @ids; # returns 14
```

upper_bound has a complexity of O(log n).

equal_range BLOCK LIST

Returns a pair of indices containing the lower_bound and the upper_bound.

Operations on sorted Lists

binsert BLOCK ITEM LIST

bsearch_insert BLOCK ITEM LIST

Performs a binary search on LIST which must be a sorted list of values. BLOCK must return

a negative value if the current element (stored in $_) is smaller, a positive value if it

is bigger and zero if it matches.

ITEM is inserted at the index where the ITEM should be placed (based on above search).

That means, it's inserted before the next bigger element.

```
  @l = (2,3,5,7);
  binsert { $_ <=> 4 }  4, @l; # @l = (2,3,4,5,7)
  binsert { $_ <=> 6 } 42, @l; # @l = (2,3,4,42,7)
```

You take care that the inserted element matches the compare result.

bremove BLOCK LIST

bsearch_remove BLOCK LIST

Performs a binary search on LIST which must be a sorted list of values. BLOCK must return

a negative value if the current element (stored in $_) is smaller, a positive value if it

is bigger and zero if it matches.

The item at the found position is removed and returned.

```
  @l = (2,3,4,5,7);
```

bremove { $_ <=> 4 }, @l; # @l = (2,3,5,7);

Counting and calculation

true BLOCK LIST

Counts the number of elements in LIST for which the criterion in BLOCK is true. Sets $_ for each item in LIST in turn:

printf "%i item(s) are defined", true { defined($_) } @list;

false BLOCK LIST

Counts the number of elements in LIST for which the criterion in BLOCK is false. Sets $_ for each item in LIST in turn:

printf "%i item(s) are not defined", false { defined($_) } @list;

reduce_0 BLOCK LIST

Reduce LIST by calling BLOCK in scalar context for each element of LIST. $a contains the progressional result and is initialized with 0. $b contains the current processed element of LIST and $_ contains the index of the element in $b.

The idea behind reduce_0 is summation (addition of a sequence of numbers).

reduce_1 BLOCK LIST

Reduce LIST by calling BLOCK in scalar context for each element of LIST. $a contains the progressional result and is initialized with 1. $b contains the current processed element of LIST and $_ contains the index of the element in $b.

The idea behind reduce_1 is product of a sequence of numbers.

reduce_u BLOCK LIST

Reduce LIST by calling BLOCK in scalar context for each element of LIST. $a contains the progressional result and is uninitialized. $b contains the current processed element of LIST and $_ contains the index of the element in $b.

This function has been added if one might need the extra of the index value but need an individual initialization.

Use with caution: In most cases "reduce" in List::Util will do the job better.

minmax LIST

Calculates the minimum and maximum of LIST and returns a two element list with the first element being the minimum and the second the maximum. Returns the empty list if LIST was empty.

The "minmax" algorithm differs from a naive iteration over the list where each element is compared to two values being the so far calculated min and max value in that it only

requires 3n/2 - 2 comparisons. Thus it is the most efficient possible algorithm.

However, the Perl implementation of it has some overhead simply due to the fact that there are more lines of Perl code involved. Therefore, LIST needs to be fairly big in order for "minmax" to win over a naive implementation. This limitation does not apply to the XS version.

minmaxstr LIST

Computes the minimum and maximum of LIST using string compare and returns a two element list with the first element being the minimum and the second the maximum. Returns the empty list if LIST was empty.

The implementation is similar to "minmax".

ENVIRONMENT

When "LIST_MOREUTILS_PP" is set, the module will always use the pure-Perl implementation and not the XS one. This environment variable is really just there for the test-suite to force testing the Perl implementation, and possibly for reporting of bugs. I don't see any reason to use it in a production environment.

MAINTENANCE

The maintenance goal is to preserve the documented semantics of the API; bug fixes that bring actual behavior in line with semantics are allowed. New API functions may be added over time. If a backwards incompatible change is unavoidable, we will attempt to provide support for the legacy API using the same export tag mechanism currently in place.

This module attempts to use few non-core dependencies. Non-core configuration and testing modules will be bundled when reasonable; run-time dependencies will be added only if they deliver substantial benefit.

CONTRIBUTING

While contributions are appreciated, a contribution should not cause more effort for the maintainer than the contribution itself saves (see Open Source Contribution Etiquette <http://tirania.org/blog/archive/2010/Dec-31.html>).

To get more familiar where help could be needed - see List::MoreUtils::Contributing.

BUGS

There is a problem with a bug in 5.6.x perls. It is a syntax error to write things like:

```
my @x = apply { s/foo/bar/ } qw{ foo bar baz };
```

It has to be written as either

```
my @x = apply { s/foo/bar/ } 'foo', 'bar', 'baz';
```

or

```
my @x = apply { s/foo/bar/ } my @dummy = qw/foo bar baz/;
```

Perl 5.5.x and Perl 5.8.x don't suffer from this limitation.

If you have a functionality that you could imagine being in this module, please drop me a line. This module's policy will be less strict than List::Util's when it comes to additions as it isn't a core module.

When you report bugs, it would be nice if you could additionally give me the output of your program with the environment variable "LIST_MOREUTILS_PP" set to a true value. That way I know where to look for the problem (in XS, pure-Perl or possibly both).

SUPPORT

Bugs should always be submitted via the CPAN bug tracker.

You can find documentation for this module with the perldoc command.

```
perldoc List::MoreUtils
```

You can also look for information at:

- RT: CPAN's request tracker
  <https://rt.cpan.org/Dist/Display.html?Name=List-MoreUtils>
- AnnoCPAN: Annotated CPAN documentation
  <http://annocpan.org/dist/List-MoreUtils>
- CPAN Ratings
  <http://cpanratings.perl.org/dist/List-MoreUtils>
- MetaCPAN
  <https://metacpan.org/release/List-MoreUtils>
- CPAN Search
  <http://search.cpan.org/dist/List-MoreUtils/>
- Git Repository
  <https://github.com/perl5-utils/List-MoreUtils>

Where can I go for help?

If you have a bug report, a patch or a suggestion, please open a new report ticket at CPAN (but please check previous reports first in case your issue has already been addressed) or open an issue on GitHub.

Report tickets should contain a detailed description of the bug or enhancement request and at least an easily verifiable way of reproducing the issue or fix. Patches are always welcome, too - and it's cheap to send pull-requests on GitHub. Please keep in mind that

code changes are more likely accepted when they're bundled with an approving test.

If you think you've found a bug then please read "How to Report Bugs Effectively" by Simon Tatham: <http://www.chiark.greenend.org.uk/~sgtatham/bugs.html>.

Where can I go for help with a concrete version?

Bugs and feature requests are accepted against the latest version only. To get patches for earlier versions, you need to get an agreement with a developer of your choice - who may or not report the issue and a suggested fix upstream (depends on the license you have chosen).

Business support and maintenance

Generally, in volunteered projects, there is no right for support. While every maintainer is happy to improve the provided software, spare time is limited.

For those who have a use case which requires guaranteed support, one of the maintainers should be hired or contracted. For business support you can contact Jens via his CPAN email address rehsackATcpan.org. Please keep in mind that business support is neither available for free nor are you eligible to receive any support based on the license distributed with this package.

THANKS

Tassilo von Parseval

Credits go to a number of people: Steve Purkis for giving me namespace advice and James Keenan and Terrence Branno for their effort of keeping the CPAN tidier by making List::Utils obsolete.

Brian McCauley suggested the inclusion of apply() and provided the pure-Perl implementation for it.

Eric J. Roode asked me to add all functions from his module "List::MoreUtil" into this one. With minor modifications, the pure-Perl implementations of those are by him.

The bunch of people who almost immediately pointed out the many problems with the glitchy 0.07 release (Slaven Rezic, Ron Savage, CPAN testers).

A particularly nasty memory leak was spotted by Thomas A. Lowery.

Lars Thegler made me aware of problems with older Perl versions.

Anno Siegel de-orphaned each_arrayref().

David Filmer made me aware of a problem in each_arrayref that could ultimately lead to a segfault.

Ricardo Signes suggested the inclusion of part() and provided the Perl-implementation.

Robin Huston kindly fixed a bug in perl's MULTICALL API to make the XS-implementation of part() work.

Jens Rehsack

Credits goes to all people contributing feedback during the v0.400 development releases.

Special thanks goes to David Golden who spent a lot of effort to develop a design to support current state of CPAN as well as ancient software somewhere in the dark. He also contributed a lot of patches to refactor the API frontend to welcome any user of List::MoreUtils - from ancient past to recently last used.

Toby Inkster provided a lot of useful feedback for sane importer code and was a nice sounding board for API discussions.

Peter Rabbitson provided a sane git repository setup containing entire package history.

TODO

A pile of requests from other people is still pending further processing in my mailbox.

This includes:

? delete_index

? random_item

? random_item_delete_index

? list_diff_hash

? list_diff_inboth

? list_diff_infirst

? list_diff_insecond

These were all suggested by Dan Muey.

? listify

Always return a flat list when either a simple scalar value was passed or an array-reference. Suggested by Mark Summersault.

SEE ALSO

List::Util, List::AllUtils, List::UtilsBy

AUTHOR

Jens Rehsack <rehsack AT cpan.org>

Adam Kennedy <adamk@cpan.org>

Tassilo von Parseval <tassilo.von.parseval@rwth-aachen.de>

COPYRIGHT AND LICENSE